

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF

全面讲解深度卷积网络的技术理论、工作原理、
实践方法、架构技巧和训练策略

Deep Convolutional Neural Network
Principle and Practice

深度卷积网络 原理与实践

彭博 著



机械工业出版社
China Machine Press

内容简介

深度卷积网络（DCNN）是目前十分流行的深度神经网络架构，它构造清晰直观，效果引人入胜，在图像、视频、语音、语言领域都有广泛应用。

本书以AI领域最新的技术研究和实践为基础，从技术理论、工作原理、实践方法、架构技巧、训练方法、技术前瞻等6个维度对深度卷积网络进行了系统、深入、详细的讲解。

以实战为导向，深入分析AlphaGo和GAN的实现过程、技术原理、训练方法和应用细节，为读者依次揭开神经网络、卷积网络和深度卷积网络的神秘面纱，让读者了解AI的“思考过程”，以及与人类思维的相同和不同之处。

本书在逻辑上分为3个部分：

第一部分 综述篇（第1、6、9章）

这3章不需要读者具备编程和数学基础，对深度学习和神经网络的基础知识、AlphaGo的架构设计和工作原理，以及深度学习和人工智能未来的技术发展趋势进行了宏观介绍。

第二部分 深度卷积网络篇（第2、3、4、5章）

结合作者的实际工作经验和案例代码，对深度卷积网络的技术理论、工作原理、实践方法、架构技巧和训练方法做了系统而深入的讲解。

第三部分 实战篇（第7、8章）

详细分析了AlphaGo和GAN的技术原理、训练方法和应用细节，包括详细的代码分析和大量GAN的精彩实例。

本书的案例代码在GitHub上提供下载，同时读者

可在GitHub上与作者交流与本书相关的问题。

Deep Convolutional Neural Network
Principle and Practice

深度卷积网络 原理与实践

彭博 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

深度卷积网络：原理与实践 / 彭博著. —北京：机械工业出版社，2018.3
(智能系统与技术丛书)

ISBN 978-7-111-59665-3

I. 深… II. 彭… III. 人工神经网络—研究 IV. TP183

中国版本图书馆 CIP 数据核字 (2018) 第 064727 号

深度卷积网络：原理与实践

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：李 艺

责任校对：李秋荣

印 刷：北京市兆成印刷有限责任公司

版 次：2018 年 5 月第 1 版第 1 次印刷

开 本：186mm×240mm 1/16

印 张：20.5

书 号：ISBN 978-7-111-59665-3

定 价：129.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88379426 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

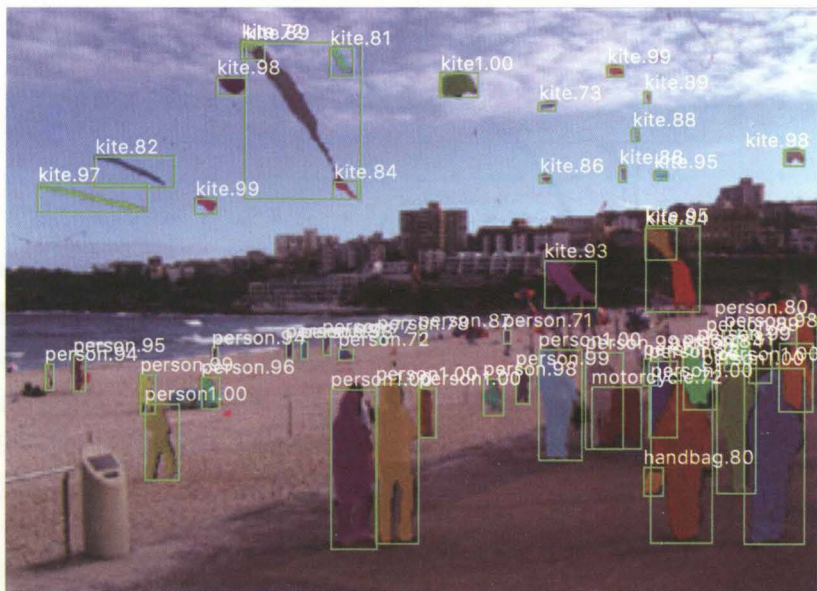
Preface 前言

为何写作本书

自 2012 年以来，随着深度学习（Deep Learning, DL）的快速发展，人工智能（Artificial Intelligence, AI）取得了长足的进展。

从语音助手、人脸识别、照片美化，到自动驾驶、医疗诊断、机器翻译，基于深度神经网络（Deep Neural Network, DNN）的新一代人工智能，已在各个领域进入我们的日常生活。许多学者认为，人工智能将开启第四次工业革命，并对人类的未来产生深远影响。

例如，通过 Mask R-CNN 深度神经网络[⊖]，电脑可快速自动识别出图像中的各个物体，用色彩和方框标记。这对于自动驾驶和机器人技术有重要意义，也是传统 AI 方法难以实现的。



⊖ 地址为 <https://arxiv.org/abs/1703.06870>。

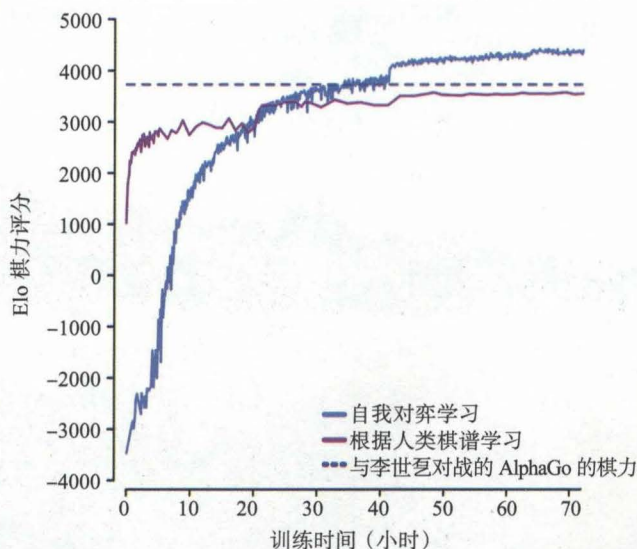
值此变革之际，我们理应跟上时代的步伐，增进对 DL 与 AI 的了解。本书的目标是：

- ❑ 如果读者没有编程和数学基础，也能在阅读后体会到深度神经网络的奥妙。
- ❑ 如果读者有一定基础，就可学会用 DL 的方法解决实际问题，为从事相关的工作和研究做好准备。

具体而言，本书选取深度卷积网络（Deep Convolutional Neural Network, DCNN）作为切入点，这是目前最流行的深度神经网络架构，其构造直观易懂，效果引人入胜，在图像、视频、语音、语言领域都有广泛应用。

我们还将结合多个实例进行讲述，让读者更深入理解深度神经网络的运作。说起深度神经网络的实例，广为人知的莫过于由 Google DeepMind 研发的 AlphaGo (<https://deepmind.com/research/alphago/>):

- ❑ 在 2016 年 3 月，AlphaGo 以 4:1 战胜韩国顶尖棋手李世乭，让 AI 成为了目前最热门的话题之一。
- ❑ 在 2017 年 5 月，新版 AlphaGo 以 3:0 战胜当今世界围棋第一人——中国的柯洁。所有棋手都认同它已全面胜过人类，但它仍需要人类棋谱作为训练的前期输入。
- ❑ 在 2017 年 10 月，名为 AlphaGo Zero 的最新版 AlphaGo 已能完全脱离人类棋谱，从零开始，纯粹依靠自我探索，自我对弈，就能实现超越此前所有版本的棋力。
- ❑ 如下图所示，蓝色的 20-blocks 版 AlphaGo Zero，最初的棋力还不如人类的初学者，但它在 24 小时内就能赶上红色的学习人类棋谱的 AlphaGo，并在 40 小时内超越与李世乭对战的 AlphaGo。

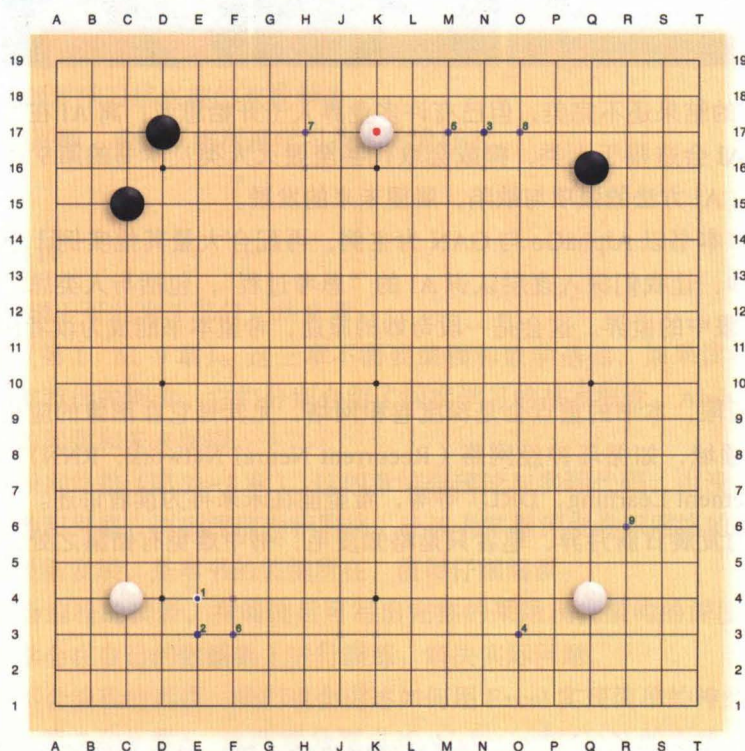


- ❑ 在 2017 年 12 月，DeepMind 还将 AlphaGo Zero 的方法用于国际象棋、日本将棋，称为 AlphaZero。它仅需几个小时的训练，就打败了此前世界最强的程序，这证明

AlphaGo 方法的通用性极强。

AlphaGo 的核心是深度卷积网络。深度神经网络的强大，关键在于能模拟人类的直觉。AlphaGo 的强大，关键在于通过深度卷积网络，成功模拟了人类的棋感。

那么，深度卷积网络是如何学会下围棋的？AlphaGo 真的理解围棋吗？AlphaGo 与人类下棋的思维有怎样的相同和不同之处？我们将在第 6 章阐述 AlphaGo 的运作，并在第 7 章亲手训练 AlphaGo 的策略网络（policy network）。如下图所示，棋盘中的标记 1 到 9，代表此时策略网络对于下一手的前 9 位推荐。

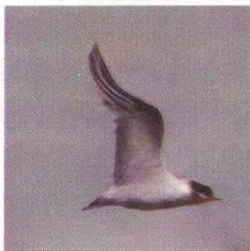


深度神经网络的威力，还不仅止于此。近年来，深度学习中热门的话题是一种新的深度神经网络范式：生成式对抗网络（Generative Adversarial Networks, GAN）。它同样基于深度卷积网络，我们会在第 8 章讲述。

GAN 试图模拟人类的更神秘之处：创造力和想象力，如自动作画、自动作曲，甚至自动发现治疗癌症的药物结构。例如，通过 StackGAN 网络[⊖]，电脑可根据人提供的文字描述，自动绘制出无穷无尽的符合描述的图像。

⊖ 地址为 <https://arxiv.org/pdf/1612.03242.pdf>。

这是一只白鸟，在头和翅膀上有些黑色，有一个长的橙色喙。



这只鸟有黄色的腹部和腿，灰色的背部和翅膀，棕色的喉部和脖子，黑色的面部。



这朵花有重叠着的粉红色尖状花瓣，围着一圈由短的黄丝构造的花蕊。



目前 GAN 的结果还不完美，但已有许多业界人士开始思考：离 AI 在所有领域超越人类还有多远？AI 会造福于人类，抑或会取代甚至毁灭人类？本书的第 9 章将讨论这一话题，并介绍当前 AI 方法的强项与缺陷，展望未来的发展。

总而言之，本书以 AlphaGo 与 GAN 为主例，再配合大量其他实例，为读者揭开深度卷积网络的面纱，让我们深入逐层认识 AI 的“思考过程”，包括与人类思维的相同和不同之处，体验 AI 眼中的世界。这会是一段奇妙的旅途，希望本书能成为读者探索深度学习世界的助手。

憾于篇幅所限，本书的重点会是深度卷积网络，尤其是它在图像的应用。对于深度学习涉及的其他领域，如循环神经网络（Recurrent Neural Network, RNN）、深度强化学习（Deep Reinforcement Learning, DRL）等等，希望能在未来再为读者讲述。

深度学习的发展日新月异，笔者只是略知皮毛，书中难免有错漏之处，恳请读者不吝指正。

本书的特点

市面已有不少介绍深度学习与深度神经网络的书籍，但多为编著或译文。本书的特点是：

- ❑ 叙述与代码范例皆会结合笔者的实际经验，如调参经验和网络架构经验，让读者掌握真正具有实用性的技巧。
- ❑ 包括深度卷积网络和 AI 的重要最新发展，如 DenseNet、Xception、各种 GAN 变种、预测学习（predictive learning）、Capsule 等。书中的许多例子都来自于 2017 年的最新研究。
- ❑ 对于重要的理论知识，如反向传播（Back Propagation, BP）的推导，本书不会回避，会详细说明。这里的细节是常见的面试题，如果读者还没有理解清楚，阅读本书就对了。

- 本书的行文力求通俗易懂，不会过于抽象。如果读者不熟悉数学，可跳过书中数学推导的部分，因为目前的深度学习框架已很完善，即使不了解数学，同样可以成功训练和使用。

本书读者对象

- 对 AI、深度学习感兴趣的开发者。
- 希望通过深度学习方法，解决实际问题的工程师。
- 希望从事 AI、深度学习相关工作的求职者。
- 对 AI、深度学习感兴趣的院校师生。
- 对 AI 感兴趣，希望了解深度学习技术的爱好者。

如何阅读本书

本书正文逻辑上可分为 3 部分，共 9 章：

- 综述篇（第 1、6、9 章）。这三章不需要编程和数学基础，如果读者尚不熟悉相关技术，推荐优先阅读。它们分别介绍了深度学习的基本概念、AlphaGo 的架构、深度学习的问题和未来展望。
- 深度卷积网络篇（第 2~5 章）。这四章结合理论与实际代码，由浅入深，从神经网络，到卷积网络，再到深度卷积网络，让读者掌握深度卷积网络的基础知识、实践技巧和最新发展，是本书的关键所在，值得仔细阅读。
 - 对于会编程的读者，在阅读后可写出完整的采用现代架构的深度卷积网络程序，以及学会在自己的数据集上进行训练，解决实际问题。
 - 对于不会编程的读者，我们也会讲述如何用 Excel 实现简单的神经网络模型。
- 实战篇（第 7 和 8 章）。这两章分别讲述 AlphaGo 和 GAN 的训练和应用细节，包括详细的代码分析。其中第 8 章还包括大量 GAN 的精彩实例，无须技术基础也能体会。

本书附录部分包括深度学习的网络资源列表。

本书内容的一些选择：

- 市面上有许多深度学习编程框架，如 TensorFlow、Caffe 等。本书选择 MXNet，因其训练速度快、占用资源少，且使用方便、架构清晰、易于二次开发。
- 在编程语言方面，目前深度学习最流行的语言是 Python，本书也会选用 Python。有时我们需要讨论理论，这就离不开数学。本书数学符号的细节如下：
- \log 都代表自然对数，即以 $e=2.71828\cdots$ 为底的对数。有的书籍会将此写成 \ln 。

- MSE 损失采用 $(X-Y)^2$ 定义。有的书籍会采用 $\frac{1}{2}(X-Y)^2$ 定义。如果读者还不知道 MSE 损失，可阅读第 2 章。

资源和勘误

本书的所有代码均存放于 <https://github.com/BlinkDL>，其中也会有本书的勘误与问题解答（例如，如果 Python 对中文 UTF-8 报错，解决方法在其中），以及读者的交流方式。如您发现书中的错漏之处，或遇到不清楚的地方，或有其他宝贵意见，请在此处告知笔者，笔者不胜感激。

目前知乎（<https://www.zhihu.com>）是中文网络上较为专业的交流平台，读者可在其中找到深度学习和 AI 的介绍和综述、对于最新论文的分析及相关问题的解答。读者可关笔者的知乎主页（<https://www.zhihu.com/people/bopengbopeng>），笔者会在知乎主页定期更新与 AI 相关的内容。

致谢

感谢出版社杨福川编辑和小艺编辑对本书写作和审校的大力支持。感谢设计师 PiPi 对本书插画的绘制。感谢家人和朋友对书稿的校对和建议。感谢知乎网友五柳希安、王佑、Simon 对本书内容的建议。

Contents 目 录

前言	
引子·神之一手	1
第 1 章 走进深度学习的世界	5
1.1 从人工智能到深度学习	5
1.2 深度神经网络的威力： 以 AlphaGo 为例	8
1.2.1 策略网络简述	9
1.2.2 泛化：看棋谱就能学会下围棋	11
1.2.3 拟合与过拟合	11
1.2.4 神经网络的速度优势	12
1.3 神经网络的应用大观	13
1.3.1 图像分类问题的难度所在	13
1.3.2 用神经网络理解图像	15
1.3.3 AlphaGo 中的神经网络	17
1.3.4 自动发现规律：从数据 A 到答案 B	17
1.3.5 神经网络的更多应用	18
1.3.6 从分而治之，到端对端学习	24
1.4 亲自体验神经网络	25
1.4.1 TensorFlow 游乐场	25
1.4.2 MNIST 数字识别实例： LeNet-5	27
1.4.3 策略网络实例	28
1.4.4 简笔画：Sketch-RNN	29
1.4.5 用 GAN 生成动漫头像	30
1.5 神经网络的基本特点	31
1.5.1 两大助力：算力、数据	31
1.5.2 从特征工程，到逐层抽象	32
1.5.3 神经网络学会的是什么	35
1.6 人工智能与神经网络的历史	36
1.6.1 人工智能的两大流派： 逻辑与统计	37
1.6.2 人工智能与神经网络的 现代编年史	37
第 2 章 深度卷积网络：第一课	42
2.1 神经元：运作和训练	43
2.1.1 运作：从实例说明	43
2.1.2 训练：梯度下降的思想	44
2.1.3 训练：梯度下降的公式	46
2.1.4 训练：找大小问题的初次尝试	48
2.1.5 训练：Excel 的实现	50
2.1.6 重要知识：批大小、mini-batch、 epoch	51
2.2 深度学习框架 MXNet：安装和使用	51

2.2.1 计算图：动态与静态	52	3.1.5 训练的障碍：欠拟合、 过拟合	82
2.2.2 安装 MXNet：准备工作	53	3.1.6 训练的细节：局部极值点、 鞍点、梯度下降算法	83
2.2.3 在 Windows 下安装 MXNet	54	3.2 神经网络的正则化	85
2.2.4 在 macOS 下安装 MXNet： CPU 版	57	3.2.1 修改损失函数：L2 和 L1 正则化	85
2.2.5 在 macOS 下安装 MXNet： GPU 版	58	3.2.2 修改网络架构：Dropout 正则化	86
2.2.6 在 Linux 下安装 MXNet	59	3.2.3 更多技巧：集合、多任务学习、 参数共享等	86
2.2.7 安装 Jupyter 演算本	59	3.2.4 数据增强与预处理	88
2.2.8 实例：在 MXNet 训练神经元 并体验调参	60	3.3 神经网络的调参	89
2.3 神经网络：运作和训练	63	3.3.1 学习速率	89
2.3.1 运作：前向传播，与非线性 激活的必要性	63	3.3.2 批大小	90
2.3.2 运作：非线性激活	64	3.3.3 初始化方法	92
2.3.3 训练：梯度的计算公式	66	3.3.4 调参实战：重返 TensorFlow 游乐场	93
2.3.4 训练：实例	69	3.4 实例：MNIST 问题	95
2.3.5 训练：Excel 的实现	70	3.4.1 重要知识：SoftMax 层、 交叉熵损失	96
2.3.6 训练：反向传播	71	3.4.2 训练代码与网络架构	98
2.3.7 重要知识：梯度消失，梯度 爆炸	72	3.4.3 超越 MNIST：最新的 Fashion-MNIST 数据集	101
2.3.8 从几何观点理解神经网络	72	3.5 网络训练的常见 bug 和检查方法	103
2.3.9 训练：MXNet 的实现	73	3.6 网络训练性能的提高	104
第 3 章 深度卷积网络：第二课	77	第 4 章 深度卷积网络：第三课	106
3.1 重要理论知识	77	4.1 卷积网络：从实例说明	106
3.1.1 数据：训练集、验证集、 测试集	77	4.1.1 实例：找橘猫，最原始的 方法	107
3.1.2 训练：典型过程	79	4.1.2 实例：找橘猫，更好的方法	108
3.1.3 有监督学习：回归、分类、 标签、排序、Seq2Seq	79	4.1.3 实例：卷积和池化	108
3.1.4 无监督学习：聚类、降维、 自编码、生成模型、推荐	81		

4.1.4 卷积网络的运作	111	5.1.1 深度学习革命的揭幕者： AlexNet	142
4.2 运作：AlphaGo 眼中的棋盘	112	5.1.2 常用架构：VGG 系列	145
4.2.1 棋盘的编码	113	5.1.3 去掉全连接层：DarkNet 系列	147
4.2.2 最简化的策略网络	115	5.2 网络的可视化：以 AlexNet 为例	150
4.2.3 最简化的策略网络：特征层 和卷积后的结果	116	5.3 迁移学习：精调、预训练等	155
4.3 卷积神经网络：进一步了解	122	5.4 架构技巧：基本技巧	157
4.3.1 卷积核、滤波器与参数数量的 计算	122	5.4.1 感受野与缩小卷积核	157
4.3.2 运作和训练的计算	123	5.4.2 使用 1×1 卷积核	158
4.3.3 外衬与步长	124	5.4.3 批规范化	160
4.3.4 缩小图像：池化与全局池化	126	5.4.4 实例：回顾 Fashion-MNIST 问题	161
4.3.5 放大图像：转置卷积	127	5.4.5 实例：训练 CIFAR-10 模型	164
4.4 实例：用卷积网络解决 MNIST 问题	128	5.5 架构技巧：残差网络与通道组合	169
4.4.1 网络架构的定义与参数数量的 计算	129	5.5.1 残差网络：ResNet 的思想	169
4.4.2 训练 MNIST 网络	130	5.5.2 残差网络：架构细节	171
4.4.3 在 MXNet 运行训练后的网络	131	5.5.3 残差网络：来自于集合的理解 与随机深度	172
4.4.4 调参实例	133	5.5.4 残差网络：MXNet 实现， 以策略网络为例	173
4.4.5 在 Fashion-MNIST 数据集的 结果	133	5.5.5 通道组合：Inception 模组	174
4.5 MXNet 的使用技巧	134	5.5.6 通道组合：Xception 架构， 深度可分卷积	177
4.5.1 快速定义多个层	134	5.5.7 实例：再次训练 CIFAR-10 模型	178
4.5.2 网络的保存与读取	135	5.6 架构技巧：更多进展	181
4.5.3 图像数据的打包和载入	135	5.6.1 残差网络进展：ResNext、 Pyramid Net、DenseNet	181
4.5.4 深入 MXNet 训练细节	136	5.6.2 压缩网络：SqueezeNet、 MobileNet、ShuffleNet	183
4.5.5 在浏览器和移动设备运行 神经网络	139	5.6.3 卷积核的变形	188
第 5 章 深度卷积网络：第四课	141	5.7 物体检测与图像分割	189
5.1 经典的深度卷积网络架构	142		

5.7.1 YOLO v1: 实时的物体检测网络	190	7.1.1 棋谱数据	225
5.7.2 YOLO v2: 更快、更强	192	7.1.2 落子模拟	226
5.7.3 Faster R-CNN: 准确的物体检测网络	194	7.1.3 终局判断	226
5.7.4 Mask-RCNN: 准确的图像分割网络	195	7.2 训练代码	227
5.8 风格转移	197	7.2.1 主程序: train.py	227
第 6 章 AlphaGo 架构综述	200	7.2.2 训练参数: config.py	233
6.1 从 AlphaGo 到 AlphaZero	201	7.2.3 辅助函数: util.py	234
6.1.1 AlphaGo v13 与 AlphaGo v18	201	7.2.4 棋盘随机变换: symmetry.py	235
6.1.2 AlphaGo Master 与 AlphaZero	202	7.2.5 训练实例	236
6.1.3 解决一切棋类: AlphaZero	204	7.3 对弈实战	237
6.2 AlphaGo 的对弈过程	205	第 8 章 生成式对抗网络: GAN	240
6.2.1 策略网络	205	8.1 GAN 的起源故事	240
6.2.2 来自人类的思路	208	8.2 GAN 的基本原理	242
6.2.3 蒙特卡洛树搜索与估值问题	209	8.2.1 生成模型: 从图像到编码, 从编码到图像	242
6.2.4 从快速走子估值到价值网络	211	8.2.2 GAN 的基本效果	243
6.2.5 从搜索树看策略与价值网络的作用	213	8.2.3 GAN 的训练方法	246
6.2.6 策略与价值网络的运作实例	215	8.3 实例: DCGAN 及训练过程	248
6.3 AlphaGo 中的深度卷积网络架构	217	8.3.1 网络架构	248
6.4 AlphaGo 的训练过程	219	8.3.2 训练代码	249
6.4.1 原版 AlphaGo: 策略梯度方法	219	8.4 GAN 的更多架构和应用	255
6.4.2 新版 AlphaGo: 从蒙特卡洛树搜索学习	220	8.4.1 图像转移: CycleGAN 系列	255
6.5 AlphaGo 方法的推广	221	8.4.2 生成高分辨率图像: nVidia 的改进	260
第 7 章 训练策略网络与实战	224	8.4.3 自动提取信息: InfoGAN	261
7.1 训练前的准备工作	224	8.4.4 更多应用	264
		8.5 更多的生成模型方法	266
		8.5.1 自编码器: 从 AE 到 VAE	266
		8.5.2 逐点生成: PixelRNN 和 PixelCNN 系列	267
		8.5.3 将 VAE 和 GAN 结合: CVAE-GAN	268

第9章 通向智能之秘	272		
9.1 计算机视觉的难度.....	272	9.4.1 超越反向传播：预测梯度 与生物模型.....	295
9.2 对抗样本，与深度网络的特点.....	276	9.4.2 超越神经网络：Capsule 与 gcForest.....	297
9.3 人工智能的挑战与机遇.....	278	9.4.3 泛化问题.....	300
9.3.1 棋类游戏电脑中的陷阱.....	278	9.5 深度学习与人工智能的展望.....	304
9.3.2 偏见、过滤气泡与道德困境.....	280	9.5.1 工程层面.....	304
9.3.3 语言的迷局.....	283	9.5.2 理论层面.....	304
9.3.4 强化学习、机器人与目标 函数.....	286	9.5.3 应用层面.....	305
9.3.5 创造力、审美与意识之谜.....	290		
9.3.6 预测学习：机器学习的前沿.....	293	跋 人工智能与我们的未来	306
9.4 深度学习的理论发展.....	295	附录 深度学习与 AI 的网络资源	310

引子·神之一手

2016年3月13日，韩国首尔四季酒店，Google DeepMind 人机围棋挑战赛第4局。当棋局进行到第77手，33岁的李世乭已然感到胜负的杠杆又一次无比沉重地压在肩头。

没有多少人想到这一天会来得如此之快。就在1年前，电脑围棋仍停留在被职业棋手轻松让上4子的地步。但在这场举世瞩目的人机大战中，前3局过后 AlphaGo 竟取得了全胜，世界仿佛一夜之间从“AlphaGo 胜一盘就是胜利”变为了“李世乭胜一盘就是奇迹”。

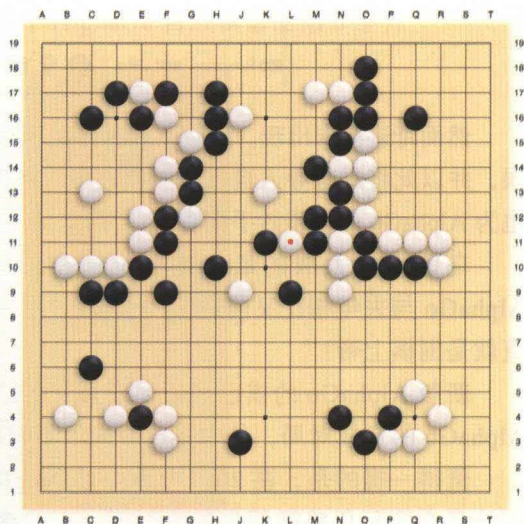
这位曾被韩国棋迷称为“不败的飞禽岛少年”，被中国棋迷称为“小李”的14次围棋世界冠军获得者，此时长考^①了整整24分钟，最终在 AlphaGo 的两颗黑子之间落下了第78手“挖”（图中用红点标注，坐标为L11）。

著名的第78手，被广誉为“神之一手”。AlphaGo 没有看见此中暗藏的杀机，不紧不慢地回应以“退”（坐标为K10），随后仿佛被击中了软肋，在战斗中溃不成军，令所有人诧异不已。

而 DeepMind 创始人 Demis Hassabis 也在 Twitter 上感慨，第78手完全出乎 AlphaGo 的意料：

- 根据 AlphaGo 策略网络的评估，人类将第78手下在L11的概率小于万分之一，这令 AlphaGo 大为“惊奇”。
- 在 AlphaGo 回应以“退”之后，AlphaGo 并未知晓自己已踏入歧途，仍认为自己稳操胜券，根据价值网络（value network）的评估，认为自己的胜率达到了70%。
- 直到第87手左右，AlphaGo 才恍然大悟，发觉了自己此前的错误，对于胜率的评估一落千丈。

如果说这一切的戏剧性还不够，经过顶尖棋手在赛后的复盘分析，这神奇的第78手，实际并不成立，只是 AlphaGo 应错了，因为如果 AlphaGo 在第79手回应以“顶”（坐标为



① 长考是围棋术语，意思是长长地思考。

2 深度卷积网络：原理与实践

L10)，李世石的获胜几率仍旧渺茫。

那么，“神之一手”为何能攻破 AlphaGo 的层层防御？AlphaGo 为何在此前的对局中如此强大？又为何会突然崩溃？AlphaGo 的策略网络和价值网络是如何运作的？希望本书能让读者找到这一切的端倪。

在随后的第 5 局中，AlphaGo 依旧显示了强大的实力，最终以 4:1 获得了世纪人机大战的胜利。

而 AlphaGo 的脚步没有停止。经过 DeepMind 的不断训练与改进，2017 年 5 月，新版 AlphaGo 在浙江乌镇重出江湖。

这一次，在 AlphaGo 面前的是当今世界围棋第一人——中国的柯洁，他时年 19 岁，但已四夺世界冠军，与李世石的交战战绩是 8 比 2 领先，长期位于世界围棋积分榜首位 (<https://www.goratings.org/zh>)，可谓是代表人类棋手的最后一位勇士。

另一方面，由于 AlphaGo 具有自我进化机制，乌镇版 AlphaGo 比此前所有版本都强大。李世石在韩国解说时认为，新版 AlphaGo 比 1 年前与自己对决时又进步了 2 个子，这在围棋中已是难以逾越的天堑。

在这次对局中，柯洁展示了第一人的风采，与 AlphaGo 激烈搏杀，特别是在第 2 局的前半盘，柯洁将棋局成功导入极其复杂的局面，棋局的质量之高，令观战者纷纷表示已看不懂两位绝世高手的招法，也令 Demis Hassabis 发 Twitter 给柯洁点赞，表示柯洁的应对可谓完美。



然而奇迹再未出现。新版 AlphaGo 的棋极其灵动，算路深远无比，足以抓住对方的任何失误，并以此将局势导入自己的控制之中，将优势牢牢保持到终盘，滴水不漏。“棋圣”聂卫平也为 AlphaGo 的一步妙手而赞叹：“阿老师（AlphaGo）的招太牛了，这个我下辈子都想不到。”

乌镇人机大战的最终比分定格在了 3:0，AlphaGo 完胜。柯洁在赛后访谈中表示，AlphaGo 与去年相比又前进了一大步：“上一次它的棋还是接近人类的，而这次 AlphaGo 简直就是围棋上帝！”

新版 AlphaGo 的棋已经近乎于真正的“神之一手”。赛后 DeepMind 团队公布了 50 局 AlphaGo 的自战对局，其中变化令人目不暇接，更被顶尖棋手称为“来自未来的棋谱”。

值得一提的是，在 2017 年 9 月，DeepMind 团队与柯洁进行了复盘，显示在第 2 局中，柯洁是在实际仍略微领先的情况下过于悲观，拼得太狠，导致了连续失误，让胜负移向了 AlphaGo。确实，人类的情绪波动既是我们的优点，也是我们的缺点。机器会失误，人类也

会失误（即使在顶级赛事中，我们也会看到棋手的失误），但机器在训练后往往能不断减少失误，人类却容易碰到瓶颈。

此时，我们必须直面的问题是：这是否意味着人类棋手在围棋领域的彻底失败，与人工智能在此领域的彻底胜利？

从竞技的角度而言，的确如此。新版 AlphaGo 依然并非完美，但它的自我进步速度飞快，进步的极限比人类远远更高，令人类望尘莫及，正如 AlphaGo Zero 的训练过程所显示的。

关于 AI，有一个著名的火车比喻：

人工智能就像一列火车，它临近时你听到了轰隆隆的声音，你在不断期待着它的到来。它终于到了，一闪而过，随后便远远地把你抛在身后。

当 200 年前，斯蒂芬森造出第一辆火车时，许多人嘲笑它没有马车快。但当火车超过马车，马车就再也追不上了。

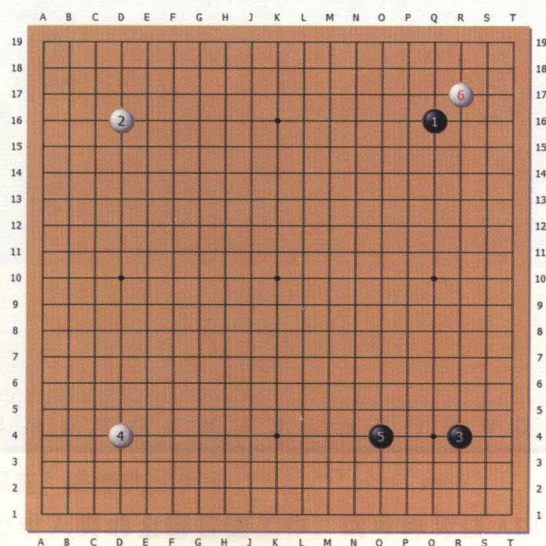
上述比喻有夸张成分，因为实际情况比这要复杂得多。但 AI 的进步速度的确有大概率会超过人类。如果 AI 在直觉、常识、逻辑思维、形象思维、连续思维、创造力等等领域全面接近人类的水准，那么离它全面超越人类就只是一纸之隔。

而一个全面超越人类智力的 AI 会如何行动？它是否会不经意间就对人类构成巨大的威胁，正如我们不经意间就会毁灭蚂蚁的巢穴并对此毫无感觉？

基于此与许多其他原因（如 AI 在军事方面的可能应用），诸多知名人士如 Elon Musk、霍金，认为我们需要对 AI 的发展加强约束。

但在另一方面，AlphaGo 离真正超级 AI 还有遥远的距离。我们仍然可将 AlphaGo 视为一种工具，而非对手。这也是目前许多研究人员的看法。

□ 首先，人脑的生物神经网络同样可通过学习而进步，人类棋手在研究 AlphaGo 的对弈棋谱时得到了许多灵感和启迪，许多 AlphaGo 的招数一跃成为流行的招法，例如在开局阶段就采用“点三三”的下法（图中 AlphaGo 自战棋谱的第 6 手）。



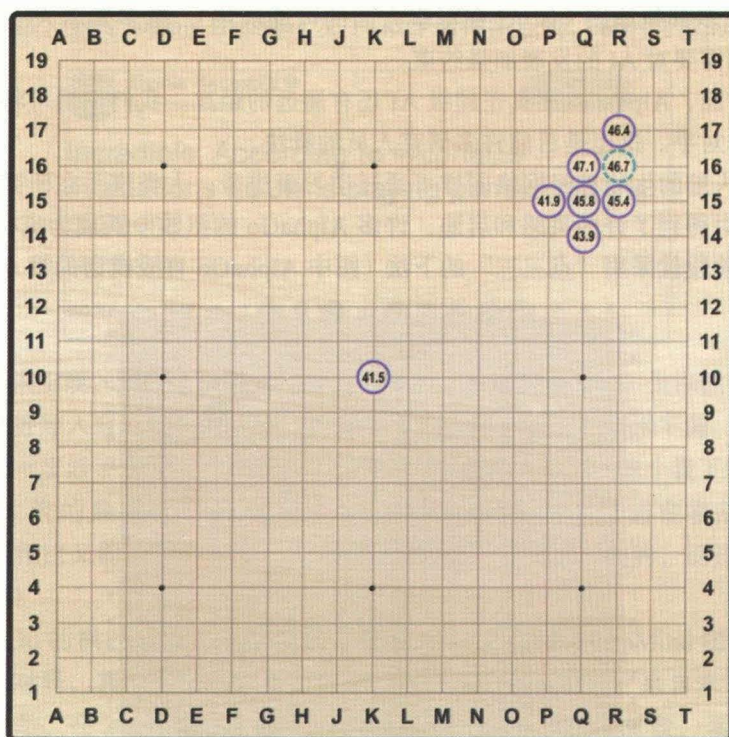
4 深度卷积网络：原理与实践

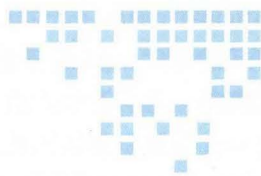
- 同时，李世乭在与 AlphaGo 赛后取得面对人类棋手的 9 连胜，柯洁更在赛前和赛后创造了 22 连胜的辉煌战绩。与 AI 的巅峰对决，无疑让棋手进入了更佳的竞技状态，提升了对于围棋的理解。
- 最后，让我们把目光转向围棋之外。AlphaGo 结合了目前 AI 中的多个领域，它的思想足以适用于各种决策问题。因此，通过对围棋的研究，促进对人类思维和决策过程的了解，解决更多的现实中的问题（如疾病诊断、蛋白质折叠、新材料开发），这是 DeepMind 开发 AlphaGo 的最重要原因。

正如 DeepMind 团队在与柯洁的对局后所言，希望 AlphaGo 能为人类探索围棋中最深远的奥秘，并发布了《AlphaGo Teach 围棋教学工具》(<https://alphagoteach.deepmind.com/>)。下图是它对于常见人类开局的黑棋胜率的预计，可见它认为黑棋略为不利，胜率相对最高的开局是下在 Q16（星位），胜率为 47.1%。

如果我们能将 AI 的发展方向控制在合理的范围，让 AI 保持为人类服务，实现人与 AI 双赢的局面，确实会是一个理想的结果。

而这种理想的结果是否一定会实现，需要我们的共同努力。如果读者在阅读本书之后，能够了解到目前 AI 的进展，对此有更多的理解，那就实现了本书写作的目的。





第 1 章

Chapter 1

走进深度学习的世界

1.1 从人工智能到深度学习

许多初次听说深度学习的朋友都会对其中各种名词的区别感到疑惑。在此我们统一进行简单介绍。

首先，从人工智能到深度学习，其演进过程如图 1-1 所示。

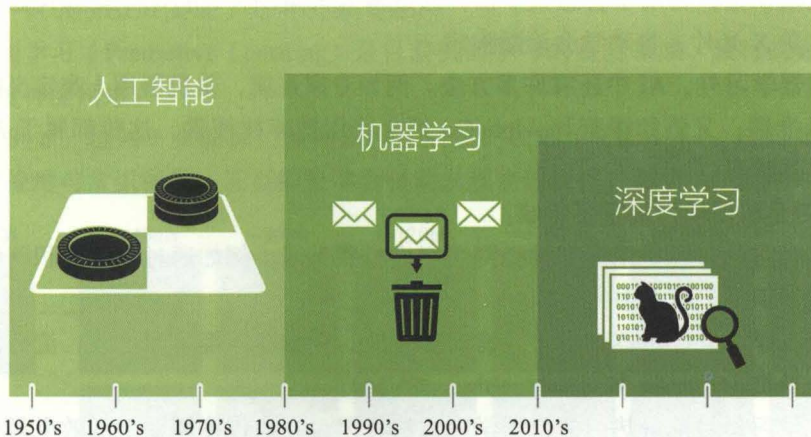


图 1-1 从人工智能到深度学习

人工智能，是最广义的概念：

□ AI 的目标是让机器完成人类的智能工作，如学习、推理、规划、创造等。从前的国际象棋程序“深蓝”是经典的例子。

○ 如果 AI 可完成人类的全部智能工作，就可称为强 AI (Strong AI) 或通用 AI

(Artificial General Intelligence, AGI)。

- 从广义的角度, AI 希望让机器完成一切人类的工作。从驾驶车辆、清扫房间到科学研究、艺术创作, AI 可能会在所有领域取代人类。
- AI 中的一大领域是理解图像, 这称为计算机视觉 (Computer Vision, CV)。CV 中的基本问题由易到难如图 1-2 所示。

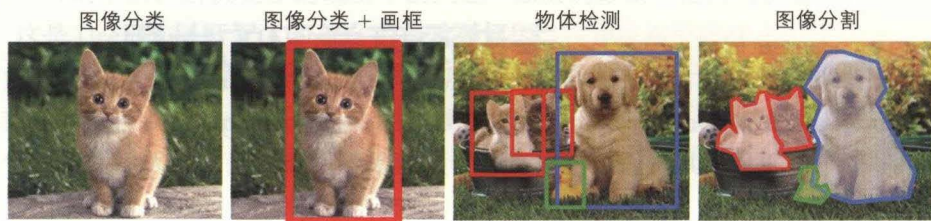


图 1-2 计算机视觉中的基本问题

- AI 中的另一大领域是理解人类语言, 这称为自然语言处理 (Natural Language Processing, NLP), 难度更高。例子包括机器翻译、语音助手等。
- NLP 可与 CV 结合, 例如, 我们可要求 AI 用语言描述图像, 看图说话, 或根据图像回答人类的问题。

机器学习 (Machine Learning) 是 AI 的重要领域:

- 机器学习的目标是让机器从训练数据中自动进步, 从经验中自动学习。例如通过输入大量棋谱, 自动学会怎么下棋; 通过输入大量画作, 自动学会如何绘画; 通过输入大量 X 光片, 自动学会诊断疾病。
- 在机器学习外, AI 中还有许多方法。例如专家系统, 是机器用人类定义的规则进行逻辑推理。又例如深蓝和 AlphaGo 都会使用博弈树搜索。这些都属于 AI, 但不属于机器学习。

深度学习是机器学习的重要领域:

- 深度学习的目标是用深度神经网络完成机器学习。例如通过深度卷积网络学习图像数据, 识别出图像中的内容。
- 在深度学习外, 机器学习中还有许多方法。例如在深度学习之前, 流行的方法是支持向量机 (Support Vector Machine, SVM)。
- 深度学习是目前最热门的机器学习方法, 因为它在许多问题上的效果最佳, 尤其是在训练数据足够多的情况下。
- 虽然深度学习非常强大, 但它并不是机器学习的全部。例如, 由卡耐基梅隆大学 (CMU) 开发的《冷扑大师》, 在 2017 年首次战胜了德州扑克的顶级职业选手, 它使用了多种机器学习方法, 但没有使用深度学习。

在机器学习中还有多个领域, 且各个领域间可相互结合:

□ 强化学习 (Reinforcement Learning) 是机器学习的另一重要领域。

- 强化学习的目标是让机器在环境中逐渐学会正确决策，最终获得最大利益。例如，如图 1-3 所示，自动学会玩游戏，在游戏结束时取得尽可能高的分数。

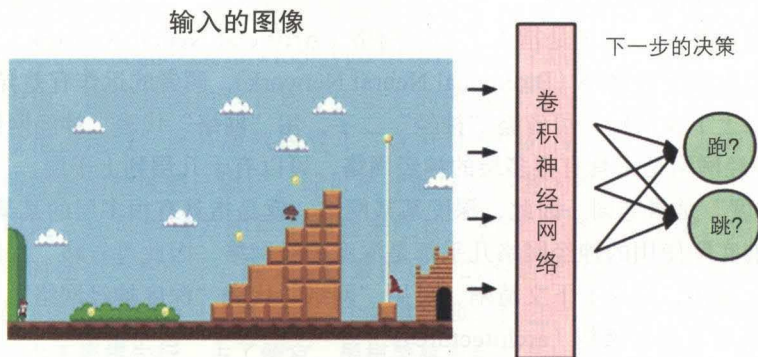


图 1-3 通过强化学习学会玩游戏

- 强化学习的难点是最终获取最大利益，这往往需要在此前做出种种牺牲。因此，不能只看眼前利益，需要逐渐建立长远的决策模型。
- 强化学习与深度学习是平行的概念，两者可结合为深度强化学习，即使用深度神经网络强化学习。这是 AlphaGo 自我进步的秘诀所在。AlphaGo 的意义深远，因为它成功结合了 AI 中的多种方法。而且目前的 AI 在大多数领域最多只能达到人类水准，但 AlphaGo 实现了完全超越人类。
- 预测学习 (Predictive Learning) 是目前机器学习的前沿领域。流行的做法也是使用深度神经网络。
- 例如，输入 1 张图像，预测图像在后续的发展 (见图 1-4)。这需要大量的逻辑推理与常识积累，甚至需要学会自动发现物理规律，对于 AI 极具挑战性。

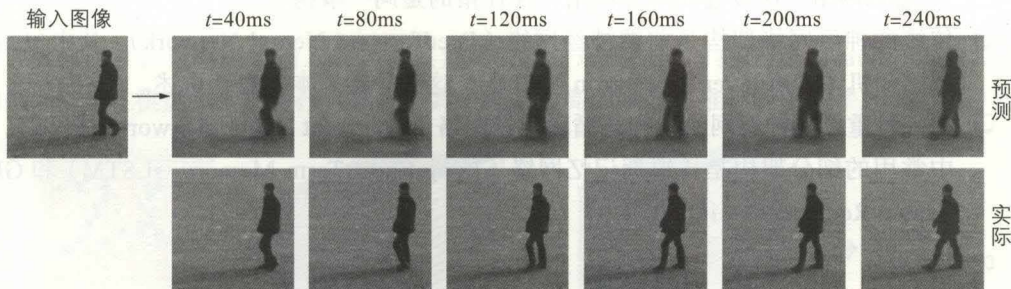


图 1-4 从输入图像预测后续发展

- 目前“深度学习三巨头”Hinton、LeCun、Bengio 均已对预测学习投入研究。笔者预测，它将可能带来 AI 的下一大飞跃。

8 ❖ 深度卷积网络：原理与实践

让我们再看各种神经网络类别。神经网络可按层级深度分为浅层神经网络与深度神经网络：

- ❑ 神经网络 (Neural Network, NN) 是由一层层的神经元 (neuron) 组成的。传统的浅层神经网络只有几层。
 - 更精确地说, 这里是指人工神经网络 (Artificial Neural Network, ANN)。因为还有生物神经网络 (Biological Neural Network), 两者的运作有差异。
 - 如果上下文清晰, 可省略“神经”二字, 用“网络”代表“神经网络”。
- ❑ 深度神经网络是指具有很多层的神经网络, 可以有十几层到上千层。
 - “深度”是修饰词。因此, 深度某某网络, 就是指具有很多层的某某网络。
 - 目前实际使用的神经网络几乎都是深度神经网络, 因此可省略“深度”二字。
 - 综上所述, 如果上下文清晰, 可用“网络”代表“深度神经网络”。

神经网络还可有多种架构 (architecture):

- ❑ 卷积神经网络 (Convolutional Neural Network, CNN) 是采用卷积 (convolution) 架构的神经网络。后文会用例子解释卷积的原理。
 - 深度卷积神经网络, 就是有很多层的卷积神经网络, 这是本书的重点。它的经典结构如图 1-5 所示, 后文会详细介绍。

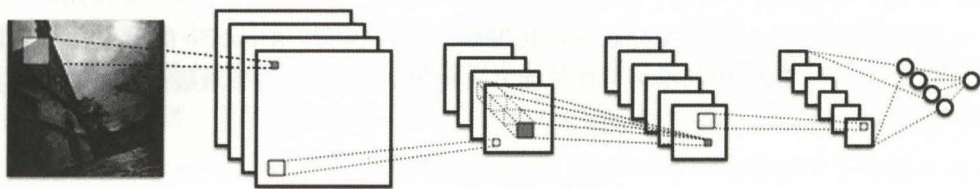


图 1-5 深度卷积网络的经典结构

- 如前所述, 可省略“深度”和“神经”。因此, 卷积网络、卷积神经网络、深度卷积网络、深度卷积神经网络, 往往指的是同一事物。
- ❑ 传统的神经网络架构是前馈神经网络 (Feedforward Neural Network), 其中包括多层感知机 (Multilayer Perceptron, MLP), 这部分知识本书也会讲述。
- ❑ 另一种重要的神经网络架构是循环神经网络 (Recurrent Neural Network, RNN), 其中常用的细分架构是长短期记忆网络 (Long-Short Term Memory, LSTM) 和 GRU (Gated Recurrent Unit)。

1.2 深度神经网络的威力：以 AlphaGo 为例

深度神经网络很神奇, 甚至有些神秘。让我们以 AlphaGo 作为实例进行讲述。

1.2.1 策略网络简述

AlphaGo 的目标是下围棋。围棋，是黑白双方轮流在 19×19 的棋盘上下子的游戏：

- 围棋在逾三千年前起源于中国，规则很简单，几分钟就能学会，变化却深奥之极，令无数人为之着迷。感兴趣的读者可参阅《PANDANET 围棋入门教程》(<http://www.weiqi-pandanet.cn/howtoplaygo/01-01.htm>) 了解入门知识。
- 简单地说，围棋是黑白双方划分棋盘上的地域的过程。在这个过程中，双方会在棋盘各处抢占地域，展开激烈的战斗，攻击对方的外围，打入对方的内部。最终，当双方所控制的地域都已稳定，即可统计地域，多者为胜。
- 下围棋时，还可将对方的棋子围起来吃掉。初学者会喜欢吃子，但高手不会贪吃，而且有时还会故意弃子，因为围棋最终比拼的不是谁吃的子多，而是谁围的地方多。
- 图 1-6 左边所示是 9×9 小棋盘的一个终盘局面，右边所示是黑白双方的地域显示。根据规则，由于黑棋先行，占了便宜，黑棋要赢得超过某一数量才算赢，称为贴目 (komi)。

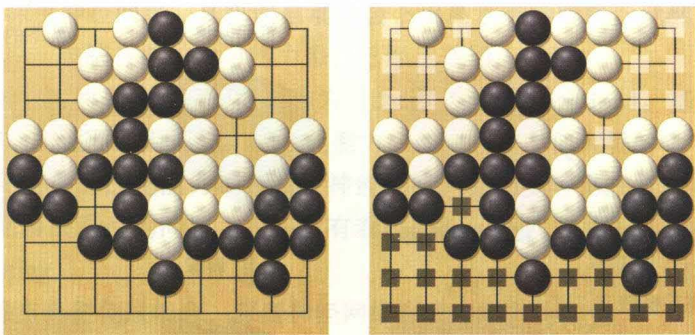


图 1-6 黑白双方在 9×9 围棋盘的地域划分实例

下围棋时，首要的问题是应该下到何处。因此，AlphaGo 的重要部件是策略网络，它是一个深度卷积网络，像一个复杂的函数，只要输入棋盘的当前局面，就可直接输出对下一手的推荐。

具体而言，策略网络输入的是将棋盘局面预处理后得到的特征层 (feature plane)，输出的是下一步在棋盘每个点的行棋概率，其中概率最高的就是最推荐的下一手，如图 1-7 所示。

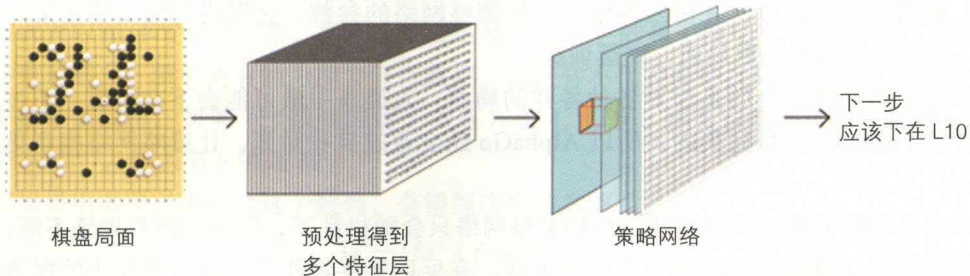


图 1-7 策略网络简图

可以想象，这个函数必定非常复杂。很难想象如何凭空写出这样的函数。AlphaGo 通过使用神经网络，解决了这个问题。

神经网络和生物的大脑神经网络有些相似，可以学习，可以不断变化：

- 神经网络具有层次化的结构，可逐层将数据进行变换。
- 神经网络的学习过程称为训练神经网络。训练过程是全自动的。
- 通过训练神经网络，它可以逐渐变成我们所需要的任何函数，包括“下围棋”函数。
 - 神经网络有许多参数，决定每层的变换方法。改变参数即可改变网络的输出。
 - 具体而言，我们会通过“随机梯度下降”（Stochastic Gradient Descent, SGD）方法不断调节参数，让网络的输出越来越接近目标。
 - 在后续章节会有直观的例子显示网络的训练过程。这个过程几乎只需要做乘法和加法，特别简单，尤其适合电脑。

例如，我们可准备大量人类高手的棋谱，然后通过训练，让策略网络学会模仿棋谱中人类高手的下法，如图 1-8 所示。

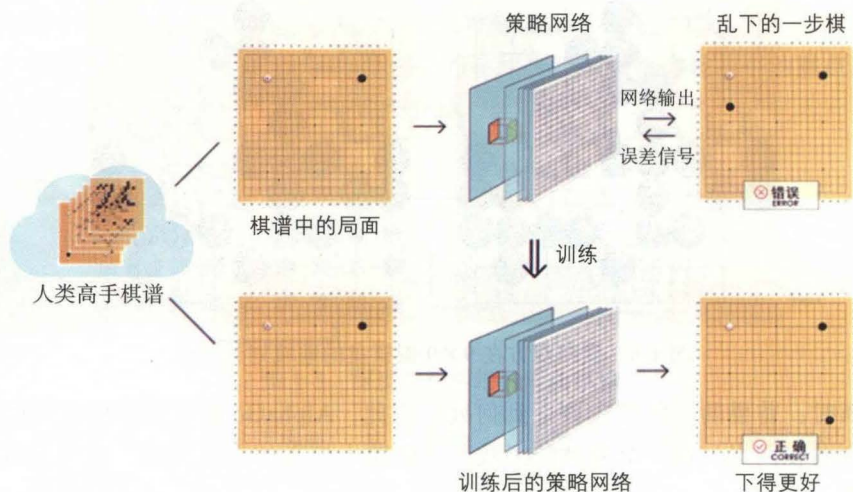


图 1-8 训练策略网络

训练的过程就是随机选取棋谱中的某个局面，将策略网络的输出与棋谱中的高手下法比较，得到“误差信号”并传入网络，然后调整网络的参数，让网络的输出更接近高手的下法。然后继续随机选取局面，继续训练。

最终策略网络学会的是面对各种各样的局面，人类高手最可能会下在哪里。不过，人类高手的下法不一定是完美的，所以 AlphaGo 还会通过自我对弈，让策略网络找到更好的下法。

策略网络的训练过程很有趣。最初策略网络只会随机乱下，但它会随着训练不断进步，下得越来越好，越来越像棋谱中高手的下法。在反反复复将几十万盘人类高手的棋谱学习多遍后，就可以下得和棋谱很像。

1.2.2 泛化：看棋谱就能学会下围棋

有经验的读者会意识到，如果电脑只是能和棋谱下得很像，其实并不稀奇。例如，我们可以写一个数据库程序，把所有棋谱都存进去，那么电脑立刻就可以下得和棋谱一模一样，但这不代表它懂围棋。

神经网络的神奇在于，它不是死记硬背棋谱，即使在面对棋谱里没有的全新的局面时，它同样可以下出颇为不错的棋。

因此，通过神经网络，我们不需要告诉电脑任何围棋的行棋道理和诀窍，只需要给电脑不断地看棋谱，它就能自动找到其中的规律，真正学会下出高水准的围棋。

这确实不可思议，这是从前的机器学习方法难以做到的。

□ 让电脑学会背诵棋谱很简单，有很多方法。

□ 让电脑通过棋谱自动学会下出高水准的围棋，就像让完全不懂围棋的人只看棋谱就无师自通，听上去就很难。如果回到10年前，大家都会认为这是个艰巨的任务。然而对于神经网络，从棋谱自动学会下出高水准的围棋，竟然就像我们吃饭喝水一样简单。电脑在此的学习能力恐怕已经比普通人更强。

神经网络的这种举一反三的能力称为“泛化”（generalization）。基于深度神经网络的AI为何如此热门？重要原因正是它的泛化能力，使它拥有类似人类的学习能力。

之前机器学习方法的泛化能力不如神经网络，它们更容易在复杂数据上出现死记硬背的现象，只能解决看过的问题，遇到没有看过的新问题就容易出现各种错误，这称为“过拟合”（over-fitting）。

机器学习的难点是避免过拟合，神经网络在此表现尤为出色。

1.2.3 拟合与过拟合

说到过拟合，让我们先了解拟合。拟合（fit）就是指和训练数据接近，例如和棋谱的下法接近。

几十年前的浅层神经网络，就足以拟合复杂的数据，这可由通用逼近定理（universal approximation theorem）证明，但它难以实际解决复杂问题，因为它容易出现过拟合。

一个著名的故事是，计算机理论之父冯·诺依曼（John von Neumann）曾说过：“用4个参数我可以拟合出一头大象，而用5个参数我可以让它的鼻子摆动”。换言之，如果模型的参数足够多，就可能有能力拟合任何数据，但这并不代表模型找到了数据背后的规律。

不妨举例说明。如果一个数列的前5项是1, 2, 4, 8, 16，那么自然的猜测是，第 x 项的公式是 $y=2^{x-1}$ ，那么第6项是32。

但如果用多项式去拟合这个数列，会得到这样的多项式：

$$y = \frac{3x^4 - 18x^3 + 69x^2 - 54x + 72}{72}$$

读者可将1, 2, 3, 4, 5分别带入此式，会发现也可分别得到1, 2, 4, 8, 16，不过，如果将

6 带入此式，会得到 31。

可见，找规律是很微妙的过程。如果读者熟悉数学，会知道第 6 项还可以等于 88888，可以等于 -3.14159 ，可以等于 0.666 ，可以等于任何数，因为不同的数都可以找到相应的多项式。我们只能说，从常理推断，正确的规律更可能是 $y=2^{x-1}$ ，第 6 项更可能是 32。

在从前的机器学习理论中，传统的观点是：

❑ 模型的参数越多，越容易过拟合。例如刚才的多项式，其中的参数很多，看上去就“不靠谱”。

❑ 模型越简单，例如刚才的 $y=2^{x-1}$ ，就越可能是正确的。

但这个观点对于深度神经网络失效了。深度神经网络的模型非常复杂，参数数目可高达几千万，甚至几亿个，却能比更简单的模型更好地克服过拟合，更可能找到正确规律。而且，即使深度神经网络出现过拟合，研究人员也可运用各种技巧改善过拟合。这部分内容后文会讲述。

为何深度神经网络拥有如此强大的威力？这仍然是学术界的研究课题。虽然已有不少论文，但迄今为止，研究人员仍然没有真正理解其中的原因。这很值得思考。这说明，我们有可能创造出我们自己也无法理解的，比我们更有智能的 AI。

目前大致的认识是，深度神经网络的逐层结构可以实现对于概念的不断抽象，这恰好与世界的运行规律吻合。自然界中有这样的逐层抽象结构，人类的大脑也具有类似的逐层抽象结构。我们在后文还会继续讨论这个问题。

1.2.4 深度神经网络的速度优势

深度神经网络的另一优势是，它的训练和运行速度很快，因为这个过程几乎只需做加法和乘法，而且非常适合并行化，后文会具体讲述。为什么这样简单的方法能有这样好的效果？这仍是未解之谜。

例如，在装载 GTX1070 显卡的家用电脑上，训练策略网络只需一两天，且为全自动，无须任何人工干预，电脑只看棋谱，不看棋书，就能达到普通成年人需要阅读棋书、请教老师、反复做题、对弈，悉心学习一两年才能达到的棋力。

如果换上神经网络专用硬件，如 Google 的 TPU 处理器，速度可再快上几十倍。如果用成千上万个 TPU，那么速度还可再提高成千上万倍。图 1-9 所示是 Google 的 TPU 深度学习系统，也正是训练 AlphaGo 所使用的系统。

这个情况很重要。因为目前常用的神经网络实际只有几百万个神经元（这里把卷积神经元看成是多个神经元，以与生物大脑的神经元连接密度一致），相当于蟑螂的水平。与之相比，猫有 7 亿个神经元，人类更有 860 亿个神经元。

根据目前的硬件发展趋势，我们将有能力使用越来越大的神经网络，神经网络的规模将在 2056 年左右达到人类的水平。届时的神经网络会拥有怎样的能力，确实不可轻视。

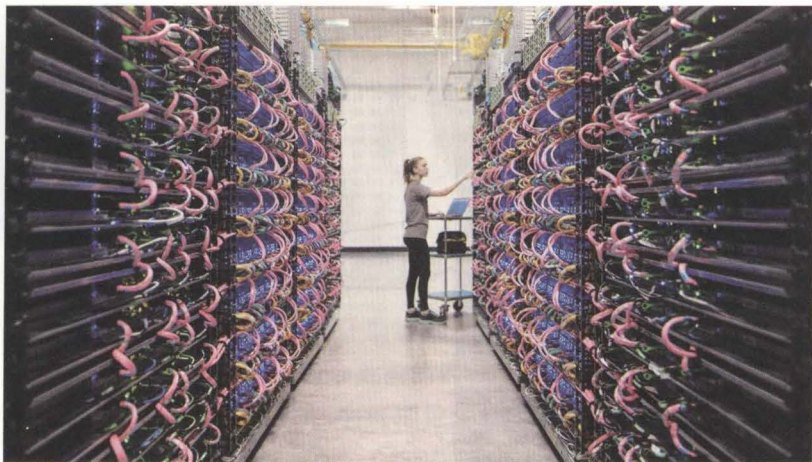


图 1-9 Google 的 TPU 服务器

1.3 深度神经网络的应用大观

关于目前的深度神经网络，有一个经验法则：如果一项工作中所需要思考和决策的问题，人能在 5 秒内解决，它就很有可能被目前的深度神经网络实现。

这已经很厉害，因为人可以在几秒之内完成相当多的工作。让我们看看深度神经网络已经可以做到什么。

1.3.1 图像分类问题的难度所在

深度学习中最经典的问题是图像分类（image classification），例如将输入的图像分类为“猫的图像”和“狗的图像”。

大家可能会感到奇怪，这真的很难吗？因为当我们看到一张图像，一眼就能看出其中的各个物体，它们的位置、大小、之间的关系，以及场景的情况。

但在深度学习取得巨大进展之前，电脑很难判断一张图像中是猫还是狗，这是因为电脑只能看到像素构成的二维数组，看到一群数字，如图 1-10 所示。

如何从一群数字中看出物体是什么，对于电脑来说很难，因为电脑在某种意义上很笨。我们用简单的例子说明电脑有多笨，教会电脑识别图像有多难。图 1-11 中的两张格点图案，在人看来是类似的，但在电脑看来区别很大。

这是因为，右图比左图向上移动了一个像素。我们可将两张图的像素数组画出来，如图 1-12 所示。

可见，数组中的大多数位置对应的值完全不同，因此电脑认为这两张图是天壤之别，因为电脑只会把两边的点一一对应，除非我们教会电脑用更好的方法观察。

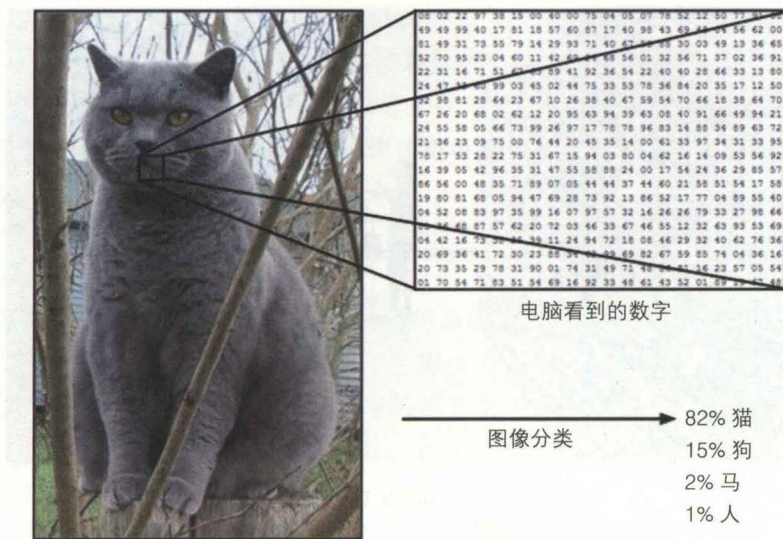


图 1-10 电脑眼中的图像是像素的数组



图 1-11 在人看来类似的格点图案



1	1	1	1	1
0	1	0	1	0
1	0	1	0	1
0	1	0	1	0
1	0	1	0	1

0	1	0	1	0
1	0	1	0	1
0	1	0	1	0
1	0	1	0	1
1	1	1	1	1

图 1-12 格点图案的像素数组

人会认为这两张图是类似的，说明人是站在更高、更整体的角度观察图像。

读者会问，是否可加入搜索，让电脑去试图移动图像，找到合适的位置？的确可以，但事情没这么简单：

- ❑ 如果我们把图像旋转一些、放大缩小一些、扭曲一些、某个局部变形一些等，也应是类似的图像。是否电脑还要一个个去实验这种种变形的方法？
- ❑ 有的物体变形之后就不像原来的物体了，有的物体却在变形之后仍然像原来的物体。电脑如何学会这一点？
- ❑ 物体的不同部位，对于不同变形的忍受程度不同，如此复杂的情况，电脑怎样能自动学会？
- ❑ 图像还可以加各种背景、各种遮挡、各种干扰，电脑该如何克服？

再看更具体的例子，如图 1-13 所示。

- ❑ 左边的狗和猫，从像素数组看很接近，因为主体都是白色，都有三个黑色小区域（眼睛、鼻子和嘴巴）。为什么一个是狗，一个是猫？
- ❑ 右边的各种狗，从像素数组看，区别很大，大小和颜色各异，为什么都是狗？



图 1-13 识别图像中的狗和猫并不是一个简单的任务

因此，计算机视觉的难点是，需要让电脑理解图像的本质和图像的语义（semantics）。传统方法在此曾遇到诸多困难，甚至曾令此前的许多研究人员感到悲观绝望。

深度神经网络彻底改变了这一切。它可把像素级别的图像逐层抽象，全自动生成语义级别的概念，简直就像变魔术。

1.3.2 用深度神经网络理解图像

在 2011 年，用传统的浅层神经网络方法，Google 动用了 16000 台机器，经过 3 天的不断计算，才构建出一个足以识别猫的网络。这在当时已被认为是了不起的成就。

在 2012 年，著名的深度神经网络 AlexNet 面世，它用 1 台机器就可轻松完成这个任务。

而在今天，通过深度神经网络，用普通的智能手机就能瞬间识别出图中的内容，而且更快更准。例如，通过名为 WhatsThatPic 的 iOS App，可轻松识别出被遮挡的橘猫（见图 1-14 所示）。

目前的深度卷积网络早已能精确分辨出上百种狗品种（见图 1-15），准确度更超人类。

它还能进一步将图像自动划分为各个物体，并标记出每个物体的具体区域，这称为图像分割（segmentation）。如图 1-16 所示，通过 Mask R-CNN 网络^①，可进一步实现包括人体姿态识别的图像分割。

深度神经网络还能自动为图像生成一段文字标题，这称为图像说明（captioning）。它还能解释选择标题中每个词的原因。图 1-17 来自 Knowing When to Look 网络^②，其中每组图像中右边的颜色可解释标题中关键词的原因。

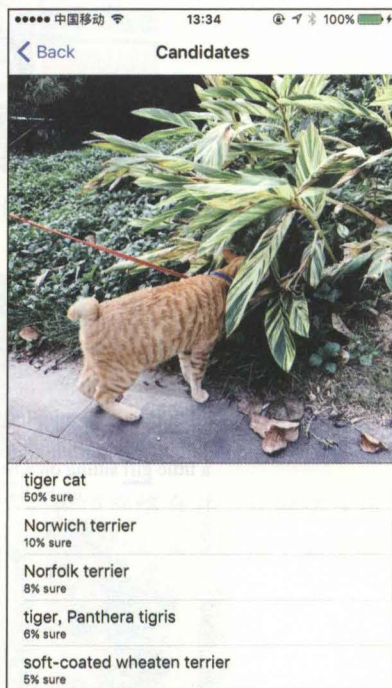


图 1-14 手机上的图像识别实例

① 地址为 <https://arxiv.org/abs/1703.06870>。

② 《Knowing When to Look》论文的地址为 <https://arxiv.org/abs/1612.01887>。



图 1-15 精确识别图中狗的品种



图 1-16 包括人体姿态识别的图像分割

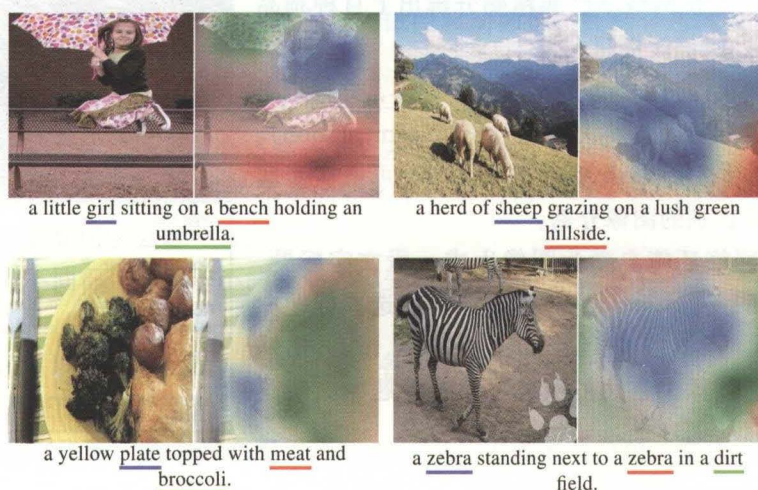


图 1-17 用深度神经网络生成图像的说明

而通过 GAN 网络,甚至可实现相反的事情,即从文字标题生成无穷无尽的相应的图像,并且同样能解释文字中的每个词对应图像中什么部分。

这很了不起,可以说深度神经网络已接近了图像的本质,只差最后的一些步骤。我们会在后文的 GAN 章节看到,它生成的图像还有一些瑕疵,来自于它在逻辑和常识上的缺陷。

这些最后的步骤是最难的,可能需要全新的方法。如果能完全解决,那么离实现真正的强人工智能就不远了。

1.3.3 AlphaGo 中的深度神经网络

AlphaGo 的策略网络实际也是在做图像分类。正如我们可以将图像分为猫、狗、人、车……策略网络实际是试图把棋盘的局面分成 361 类,对应到下一手的 361 个选点。

策略网络的成功,说明深度神经网络的分类能力确实很强。因为在围棋中,看上去类似的局面,可能存在完全不同的下一手。看上去完全不同的局面,可能存在相同的下一手。

此外,AlphaGo 还使用了价值网络。价值网络的作用是快速判断任意一个棋盘局面中自己的胜率。这实际是一个回归(regression)问题,正如我们可以从一张猫的照片判断猫的年龄,如图 1-18 所示。

在训练价值网络时,同样是将海量的棋盘局面和相应的最终胜负情况输入价值网络,它就能逐渐自动学会预测最终胜负情况。

价值网络非常重要,因为围棋的价值判断即使对于人类棋手也很难。由于 AlphaGo 在自我完成了无数对局,见过的局面远超人类,总结的规律更为准确,因此目前 AlphaGo 的价值判断已远超人类棋手,并为棋手所推崇和学习。

最后,在 AlphaGo Zero 中,研究人员发现可将策略网络和价值网络合并,让网络从棋盘局面同时输出下一手的选点和当前的胜率。这可实现更强的棋力,因为这类类似于后文会介绍的多任务学习,有利于训练出更健壮的网络。

1.3.4 自动发现规律:从数据 A 到答案 B

虽然“识别图像中的物体”与“下围棋”听上去天差地别,但对于深度神经网络而言,它们竟然是类似的:

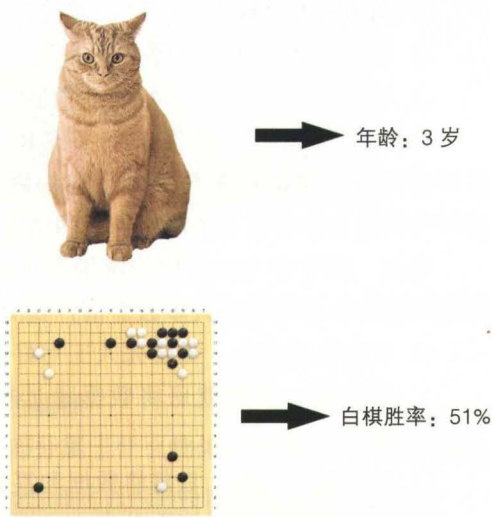


图 1-18 回归的实例

- ❑ 我们不需要告诉它人类是如何判断图像中是猫还是狗，只需要给它足够的图像分类例子（成千上万张，越多越好），它就能自动发现其中的规律，即使面对新图像，也能做出正确的判断。
- ❑ 我们不需要告诉它人类是如何判断某个棋盘局面的下一手应该在哪里，只需要给它足够的棋谱（成千上万局，越多越好），它就能自动发现其中的规律，即使面对新局面，也能做出正确的判断。

只要有充分的训练数据，并且合理地设计网络的架构和训练过程，深度神经网络就能自动学会许多复杂的问题。数据越多，越全面，训练的效果就越好。

让我们把眼光放得更远。之前提到的所有例子，都可概括为从数据 A 到答案 B：

- ❑ 图像分类：图像 → 所属类别。
- ❑ 图像分割：图像 → 每个点所属的物体和类别。
- ❑ 图像说明：图像 → 相应的标题。
- ❑ 策略网络：棋盘局面 → 下一手的位置。
- ❑ 价值网络：棋盘局面 → 胜率估计。

事实上，对于任何数据 A 与答案 B，只要有大量的 A 与 B 的对应数据，我们就能用深度神经网络自动找到 A 与 B 之间的映射关系^①。例如，在第 9 章会看到，甚至可用一个深度神经网络去学习另一个深度神经网络的训练过程，以实现更灵活高效的训练。

因此，深度学习其实比传统编程更简单，因为深度神经网络可自动完成绝大多数事情，如图 1-19 所示。

- ❑ 在传统编程中，需要先人工写好详尽的规则，然后电脑会按部就班将数据通过规则变为需要的答案。
- ❑ 在深度学习中，只需将数据和答案提供给深度神经网络，电脑就能自动自主发现其中的规则，可谓确实具有了一定的智能。



图 1-19 传统编程与深度学习的对比

基于此，已有业界人士提出“软件 2.0”“编程 2.0”的概念^②。应该说，目前的深度学习方法还不足以取代传统编程方法，但在许多场景中可以成为其重要补充和辅助。

1.3.5 深度神经网络的更多应用

深度神经网络在图像、语音、语言、视频等领域都已有丰富的应用，在此我们介绍其中的部分例子。

① 用专业术语说，这称为“有监督学习”（Supervised Learning）。此外，深度神经网络也可用于“无监督学习”（Unsupervised Learning）。后文会介绍这些概念。

② 相关的博客地址为 <https://petewarden.com/2017/11/13/deep-learning-is-eating-software/> 和 <https://medium.com/@karpathy/software-2-0-a64152b37c35>。

1. 医学：医学影像→诊断

根据 Stanford 大学的研究^①，深度卷积网络可以比医生更准确地从图像诊断皮肤癌，如图 1-20 所示。

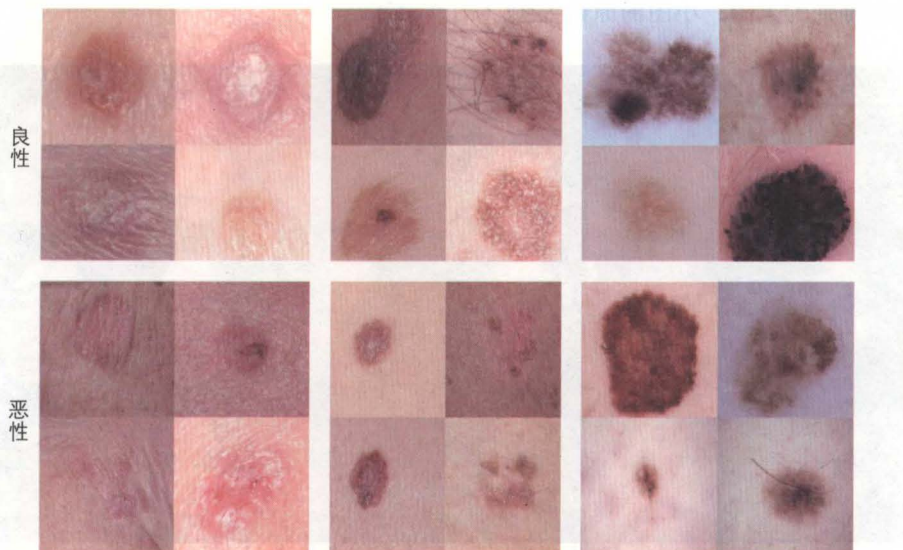


图 1-20 医学图像的识别

由图可见，皮肤的良性和恶性变异颇为相似，辨别的难度较高。类似的例子还包括从肺部 X 光片诊断肺病^②。

2. 自动上色 (colorization): 黑白图像 → 彩色图像

如图 1-21^③所示，网络可以自动理解图像不同部分属于什么物体，并加上相应的色彩。

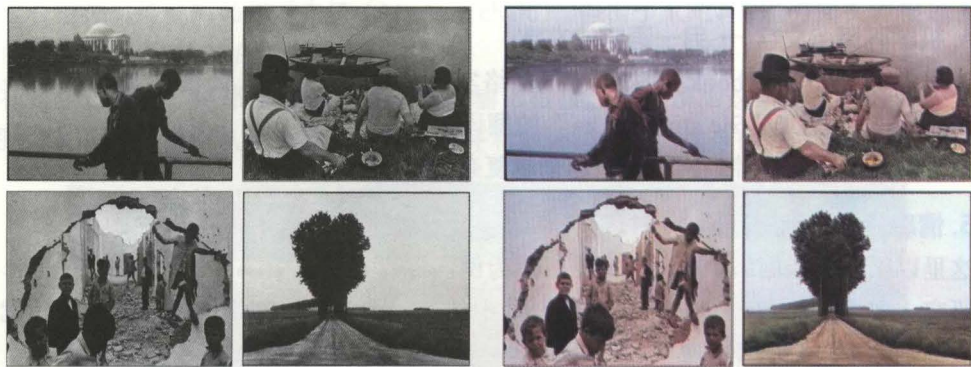


图 1-21 图像自动上色

① 地址为 <http://cs.stanford.edu/people/esteva/nature/>。

② 地址为 <https://stanfordmlgroup.github.io/projects/chexnet/>。

③ 地址为 <http://richzhang.github.io/colorization/>。

3. 超分辨率 (super-resolution): 小图→清晰大图

这里的例子是 ESPCN 网络^①，这是 SRCNN 网络的改进版，其速度很快。

图 1-22 所示中由上到下的 3 行分别为：小图直接放大（所以可以看到像素点）；经过网络处理后得到的大图；小图的原始大图。

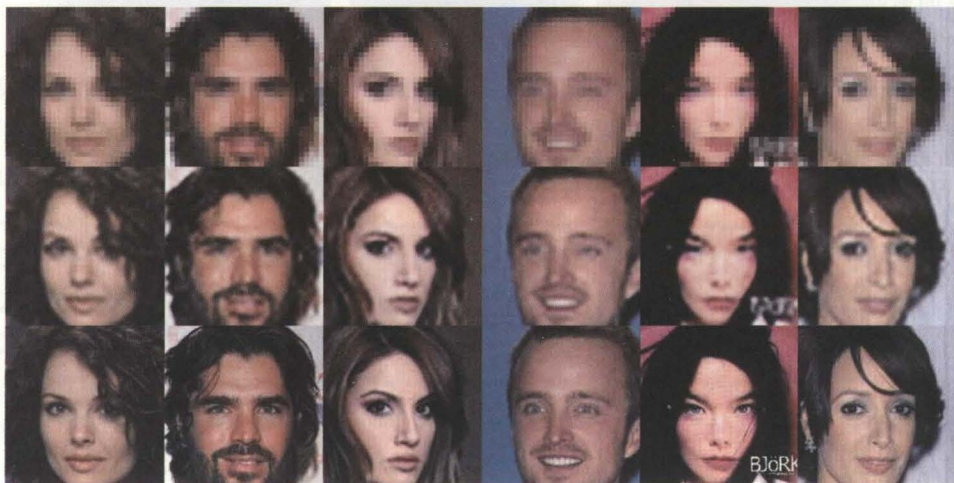


图 1-22 图像超分辨率

超分辨率的目标是从第 1 行生成第 3 行。由于缩小的过程已经丢失了信息，所以我们只能从第 1 行生成第 2 行，但效果已经不错，接近第 3 行。

使用更深层架构的 SRResNet 网络的效果更好^②，不过速度更慢。在 SRResNet 的论文中还包括了通过 GAN 方法构造的 SRGAN 网络，它的效果更清晰，但不一定接近原图，我们会在后文的 GAN 章节介绍。

4. 机器翻译：文字 → 另一语言的文字

这里的例子是 Facebook 用深度卷积网络实现的翻译模型^③，如图 1-23 所示，是英文 “They agree” 到德文 “Sie stimmen zu” 的翻译过程。

由图可见，现代的神经网络模型越来越复杂和精巧。

5. 情感分析：用户评论 → 好评度

这里以循环神经网络中的 LSTM 网络^④为例。

研究人员发现，仅仅要求网络预测评论中的下一个字母，网络中的某个神经元就会自发地准确预测评论是好评还是差评。

① 地址为 <https://github.com/Tetrachrome/subpixel>。

② 地址为 <https://arxiv.org/pdf/1609.04802.pdf>。

③ 地址为 <https://github.com/facebookresearch/fairseq>。

④ 地址为 <https://blog.openai.com/unsupervised-sentiment-neuron/>。

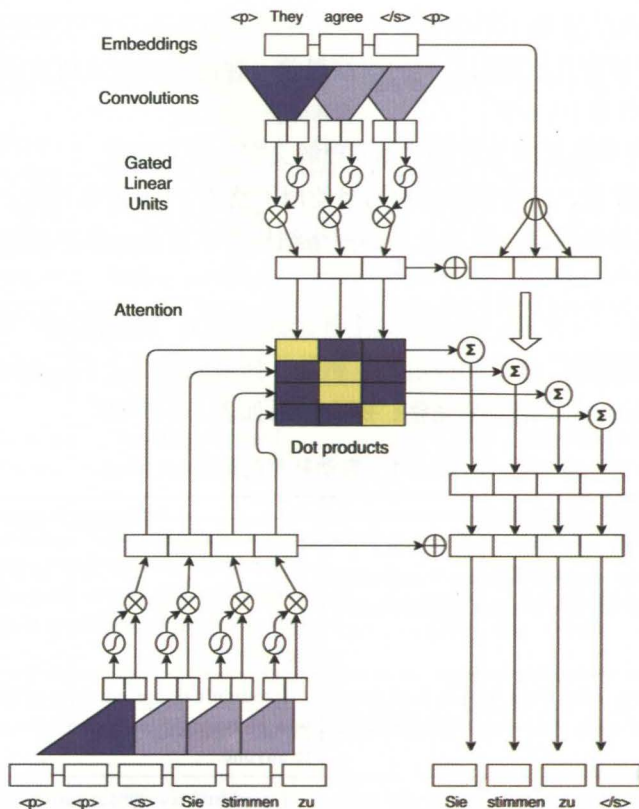


图 1-23 使用深度卷积网络的翻译模型

具体方法是，每次输入评论的一部分，然后让网络预测后续的一个字母，由于预测了一个字母后又可以预测下一个字母，最终网络可自动生成完整的评论。

这有些类似传统的马尔可夫链（Markov chain），但由于 RNN 有能力回头看很远，并做复杂的决策，所以 RNN 的效果更好。

在这个过程中，网络中的某个神经元的输出很接近于在预测这个评论是好评还是差评（见图 1-24），尽管我们从来没有告诉网络好评和差评的概念。这令研究人员很惊讶。

This is one of Crichton's best books. The characters of Karen Ross, Peter Elliot, Munro, and Amy are beautifully developed and their interactions are exciting, complex, and fast-paced throughout this impressive novel. And about 99.8 percent of that got lost in the film. Seriously, the screenplay AND the directing were horrendous and clearly done by people who could not fathom what was good about the novel. I can't fault the actors because frankly, they never had a chance to make this turkey live up to Crichton's original work. I know good novels, especially those with a science fiction edge, are hard to bring to the screen in a way that lives up to the original. But this may be the absolute worst disparity in quality between novel and screen adaptation ever. The book is really, really good. The movie is just dreadful.

图 1-24 “情感神经元”的输出

图 1-24 中显示的是这个神经元的预测，随着输入的文字越来越长，它的预测会不断变化：

- 评论的前几句在赞扬电影的原著，因此神经元也认为评论可能是好评（绿色）。
- 评论的后半部分开始指出电影的改编很差，神经元也会立刻发现这一点，将评论判定为差评（红色）。

换言之，神经网络如同是自动学会了理解文字，令人赞叹：

- 这是数据的胜利，因为训练这个网络使用了整整 8200 万条评论。
- 这是深度学习的胜利，因为这里的网络结构并不复杂，但它成功从数据中提炼出了精华。

研究人员还做了一个实验，就是将这个神经元固定为“必须好评”或“必须差评”，然后要求网络以此生成评论，效果同样出色。例如，如果固定以“I couldn't figure out”作为开头，然后让网络自动将评论补充完整，那么结果如表 1-1 所示。

表 1-1 自动补充的评论

“必须好评”生成的评论	“必须差评”生成的评论
I couldn't figure out the shape at first but it definitely does what it's meant to do. It's a great product and I recommend it highly.	I couldn't figure out how to use the product. It did not work. At least there was no quality control; this tablet does not work. I would have given it zero stars, but that was not an option.
I couldn't figure out why this movie had been discontinued! Now I can enjoy it anytime I like. So glad to have found it again.	I couldn't figure out how to set it up being that there was no warning on the box. I wouldn't recommend this to anyone.
I couldn't figure out how to use the video or the book that goes along with it, but it is such a fantastic book on how to put it into practice!	I couldn't figure out how to use the gizmo. What a waste of time and money. Might as well throw away this junk.
I couldn't figure out how to use just one and my favorite running app. I use it all the time. Good quality, You can't beat the price.	I couldn't figure out how to stop this drivel. At worst, it was going absolutely nowhere, no matter what I did. Needless to say, I skim-read the entire book. Don't waste your time.

可见，网络竟然还可以熟练地使用转折，虽然部分语句有点奇怪，但确实已经相当真实。而且，在整个过程中，我们没有教过网络任何英文语法，一切都是它自行发现的，网络在一开始甚至连英文单词都不会拼写。深度神经网络的威力确实令人惊讶。

6. 自动水军：微博 → 评论

这里的例子^①同样使用循环神经网络。它可根据用户所发微博自动生成相关评论，如图 1-25 所示。

图中左边一列是微博，右边三列是不同神经网络自动生成的评论，拟真度很高，远胜过第四列的只采用简单匹配的评论算法。也许现在不少微博用户已是这样的“智能僵尸粉”。

① 地址为 <https://arxiv.org/pdf/1503.02364.pdf>。

ID	Post	NRM-glo	NRM-loc	NRM-hyb	Rtr.-based
1	每年元旦都来高烧。 High fever attacks me every New Year's Day	多休息，增强免疫力！ Rest more to boost your immune system.	每年元旦都去医院看病 Go to see a doctor every New Year's Day	祝早日康复，身体健康！ Get well soon and stay healthy!	高烧... High fever...
2	又老了一岁，感谢我们组，我很高兴 I gain one more year. Grateful to my group, so happy	生日快乐！永远支持你！ Happy birthday! Will stand by you forever!	生日快乐！永远年轻漂亮！ Happy birthday! Wish you always stay young and pretty!	老了老了，岁月不饶人啊！ Getting old now. Time has no mercy.	我只是老了一岁 I just gain only one more year.
3	吃素第一天，坚持住，崔朵拉。 First day of being a vegetarian. Hold on, CuiDuoLa	必须来支持！加油！ I come to offer my support. Keep it up!	坚持住，坚持下去。 Hold on, keep it up.	每天都是素食主义者 I have been being a vegetarian everyday	坚持住！ Hold on!
4	这种天气真应该带个帅哥出去踏青赏花 We should go out with some cute guys to enjoy a great outing in such a nice weather.	这是哪儿啊？真美！ Where is it? It is so beautiful!	阳光明媚，心情舒畅。 Such a nice sunny day! I am in a great mood.	这种天气真不是盖的。 It is indeed a very nice weather.	文山啊出去踏青寻找灵感哈哈 WenShan, let's go out to get some inspiration. Ha! Ha!

图 1-25 微博和自动生成的评论

7. 综合应用：图像 + 问题 → 答案

这里的例子(图 1-26)来自 Google DeepMind 在 2017 年 6 月提出的 Relation Network[⊖]架构。它结合了深度卷积网络与循环神经网络，可在 CLEVR 数据集[⊖]实现超越人类的回答能力。

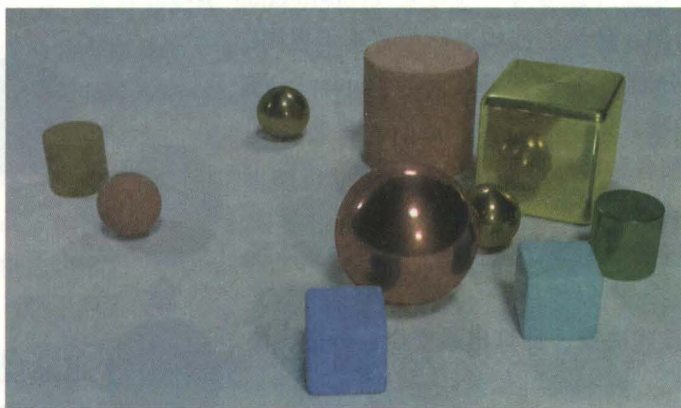


图 1-26 目标是根据此图回答问题

⊖ 地址为 <https://deepmind.com/blog/neural-approach-relational-reasoning/>。

⊖ 地址为 <http://cs.stanford.edu/people/jcjohns/clevr/>。

举例，电脑的目标是根据图 1-26 回答下列问题：

- ❑ 图中是否有相同个数的大物体和金属球？
- ❑ 在大球的左边的棕色的金属物体的左边的圆柱体有多大？
- ❑ 图中有一个与金属立方体同样尺寸的球，它与小红球的材质是否一致？
- ❑ 图中的小圆柱体与红色物体共有多少个？

可见，这些问题的逻辑关系很绕，对于人类也不容易。经多个问题测试，人类的答题准确率是 92.5%，而 DeepMind 的网络可达 95.5%，因此 DeepMind 声称在此已超越人类。

这个结果很了不起，因为通常的看法是深度神经网络不擅长逻辑推理。不过，DeepMind 的话语也有夸张成分。例如，如果我们将物体的数目大大增多，那么细心的人类仍然可以给出答案，但深度神经网络就很可能遇到困难。

在从数据 A 到答案 B，深度神经网络还有更多的成熟应用，在此列举部分例子：

- ❑ 语音识别：音频波形→文字。
- ❑ 安防：面部图像和动作姿态等→人物身份。
- ❑ 消费者金融：信用历史→还款能力。
- ❑ 广告：用户和广告的情况→用户是否会点击广告。

最后，以 GAN 为代表的生成模型，可在仅有数据的情况下，生成与数据类似的新数据，或是在多种数据中实现自动相互转换。关于此有大量精彩的实例，我们会集中在第 8 章详细介绍。

1.3.6 从分而治之，到端对端学习

深度神经网络可处理极其复杂的从 A 到 B 的情况，并且不需要训练者预先将从 A 到 B 的过程分割为各个步骤。这称为“端对端”(end-to-end)学习。

以自动驾驶为例，传统的架构可称为“基于规则”(rule-based)或“分而治之”(divide and conquer)，过程可拆分为多个步骤，如图 1-27 所示。

- 1) 从摄像头和各种传感器获得原始数据。
- 2) 提取道路、人和车辆等信息。
- 3) 配合高精度的地图等，建立一个完整的立体场景模型。
- 4) 进行逻辑推理决策，计算出最安全、舒适、快捷的车辆控制操作。
- 5) 执行车辆控制操作。回到第一步。

可以想象，这种架构需要许多人工构建和优化，人力和物力投入很大。从前的许多机器学习方法，在面对复杂问题时，也需要靠研究人员把问题预先拆分

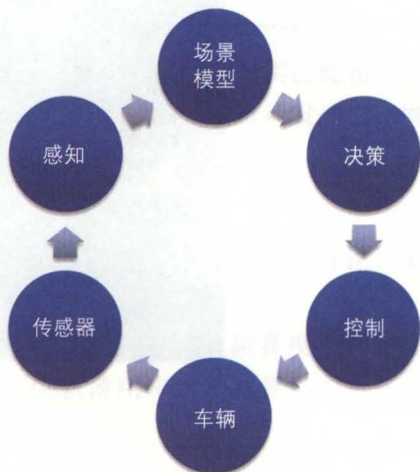


图 1-27 自动驾驶系统的拆分

成多个部分。

而端对端（end-to-end）的方法就简单、直接多了。我们会直接把摄像头看到的图像输入深度网络，要求网络直接输出对于车辆的控制指令，如方向和油门，如图 1-28 所示。

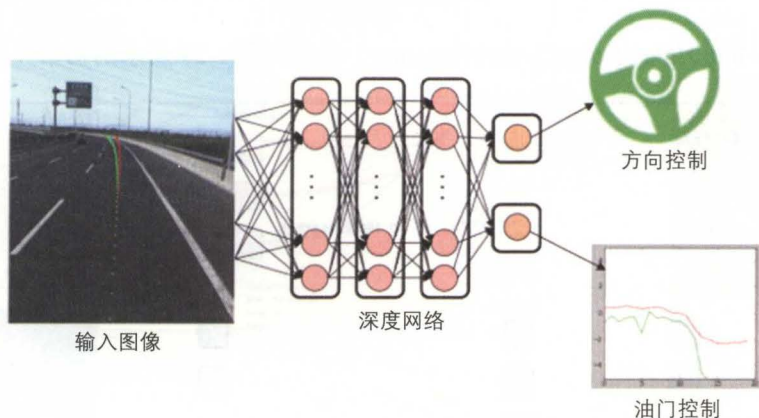


图 1-28 自动驾驶系统的端对端实现

使用端对端方法时，只要有足够的训练数据，深度网络就会自动学习，无须人类的预处理。而且训练数据可通过实地采集，很容易获得。

例如，只需给人驾驶的 vehicle 装上摄像头，记录人的方向盘和油门操作，即可获得大量训练数据。最终深度网络可学会直接从摄像头图像判断如何操作方向盘和油门，模仿人的驾驶方法。

同样，AlphaGo 的策略网络和价值网络，也可以说是端对端的。因为只需要输入棋谱，深度网络就能自动学会人类下棋的复杂思维过程。

读者也许会认为，端对端的学习方法更可能存在漏洞。但耐人寻味的是，在许多任务上，目前性能最好的方法都是端对端的。而且，端对端的方法速度更快，因为减少了许多中间环节。换言之，深度神经网络本身就拥有强大的抽象和概括能力，无须人类的预处理。

不过，纯粹端对端的方法，也会让深度神经网络变得难以理解，就像一个黑匣子。更稳妥的做法是将端对端的方法与基于规则的方法结合。两者的许多特点可以互补。

同样，AlphaGo 的深度网络，也需要和基于逻辑推理的蒙特卡洛树搜索相结合，这样才能实现最高的棋力。

1.4 亲自体验深度神经网络

在上一节，我们看到了深度神经网络的强大。本节我们将亲自体验它的运作。

1.4.1 TensorFlow 游乐场

我们先从更简单易懂的浅层神经网络看起。为了形象地演示神经网络的运作，Google

开发了称为 TensorFlow 游乐场的演示工具。

这里的 TensorFlow 是 Google 深度学习编程框架的名字。不过，TensorFlow 游乐场并未使用 TensorFlow 框架，而是直接用 Javascript 语言编写的，因为它所涉及的神经网络很简单。

打开网页 (<http://playground.tensorflow.org/>) 会看到图 1-29 所示的初始界面。

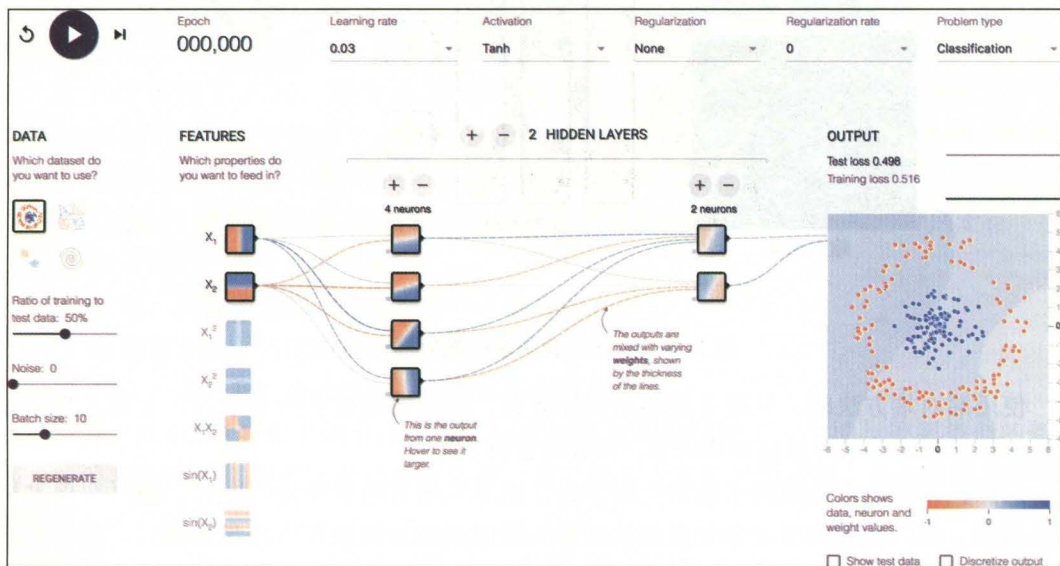


图 1-29 TensorFlow 游乐场的界面

可见神经网络类似于一个水管系统：

- ❑ 这里的输入是 2 张固定的图 x_1 和 x_2 。
- ❑ 经过 $2 \times 4 = 8$ 根水管，变成 4 张图，对应 4 个神经元的输出。
- ❑ 然后再经过 $4 \times 2 = 8$ 根水管，变成 2 张图，对应 2 个神经元的输出。
- ❑ 最后经过 $2 \times 1 = 2$ 根水管，变成最右边的 1 张大图。目标是希望这张大图能够区分黄色和蓝色的数据点。

现在点击左上角的启动按钮，启动神经网络的训练。

我们会看到各根水管开始不断变色，图也在随之不断变化，如图 1-30 所示，在一段时间后达到了目标，即将蓝色点划分到蓝色区域，将黄色点划分到黄色区域。

图中的蓝色代表正，白色代表 0，黄色代表负。系统中的每根水管都可以取一定的正负值。水管决定图之间的变换关系，具体而言，就是决定混合的权重。关于此会在后文叙述。

而神经网络的训练过程，就是不断计算出怎样最佳地调节水管的值，然后不断调节水管，让我们逐渐靠近目标。其中的具体运作，也会在后续章节详细介绍。

读者可尝试在左边选择更复杂的数据集（例如第 4 个螺旋型的数据集），以及改变水管系统的结构，例如加宽（增加每层的神经元数）和加深（增加层数）。

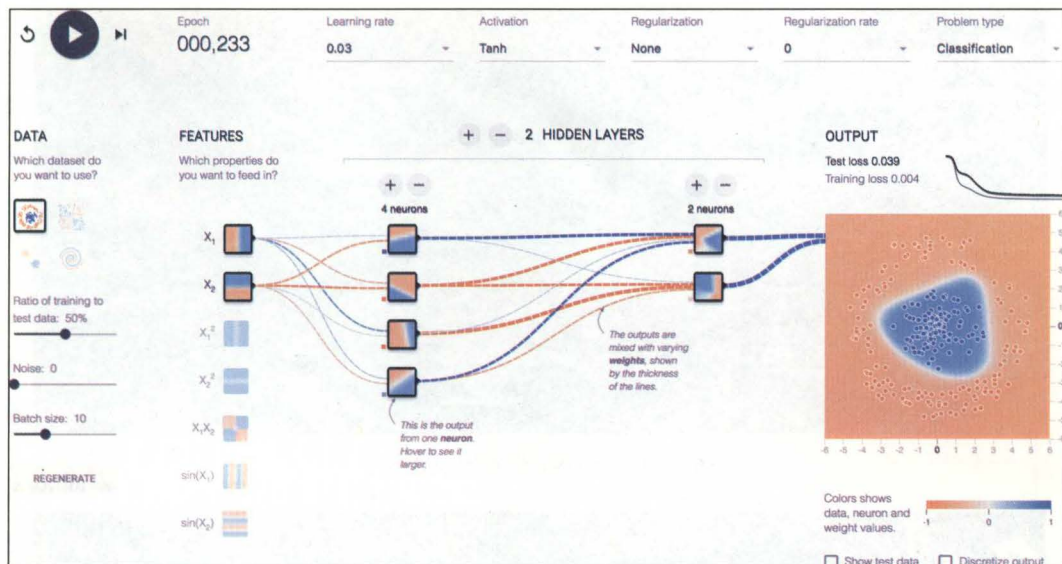


图 1-30 训练后的网络

读者会发现，水管系统越复杂，就越有能力学习复杂的数据集，但学习的过程也会越来越慢，甚至出现停滞。在后文我们会重返 TensorFlow 游乐场，进一步理解它的运作。

研究人员目前已发展出有效的技巧，可以训练极其复杂的神经网络。例如，AlphaGo v13 策略网络的输入是 48 张图，中间每一层都有 192 张图，并且有 12 个中间层。这只能算是一个普通规模的网络，因为目前复杂的网络可以有上千个中间层。

1.4.2 MNIST 数字识别实例：LeNet-5

每一本深度学习的书籍都会提到 MNIST，它是经典的数据集，目标是识别手写数字（0 到 9）。我们在后文会详细介绍，并亲手训练 MNIST 识别网络。

MNIST 的经典模型是 1998 年的 LeNet-5 网络，这是深度卷积网络的雏形，在正常情况下可达到 99.05% 的识别率。

读者可访问 <http://scs.ryerson.ca/~aharley/vis/conv/>，然后可在左上角写一个数字，并看到 LeNet-5 网络中每一层输出的图像。

请看成功识别数字 2 的例子。图 1-31 中所示网络的底层是输入图像，中间经过多层，顶层是输出。输出有 10 个小图，分别代表网络认为数字属于 0 到 9 的概率。这里的输出是正确答案 2。

但如果读者故意写一个变形大的数字，网络识别就可能失误。例如电脑可能会把变形的 2 认为是 6 或者 0，如图 1-32 所示。

这是深度卷积网络和人脑思维的不同之处。将数字做较大的变形，人仍然可轻松辨别，而深度卷积网络只能部分适应，在变形过大的时候，就会出现识别错误。

解决方法有两种：

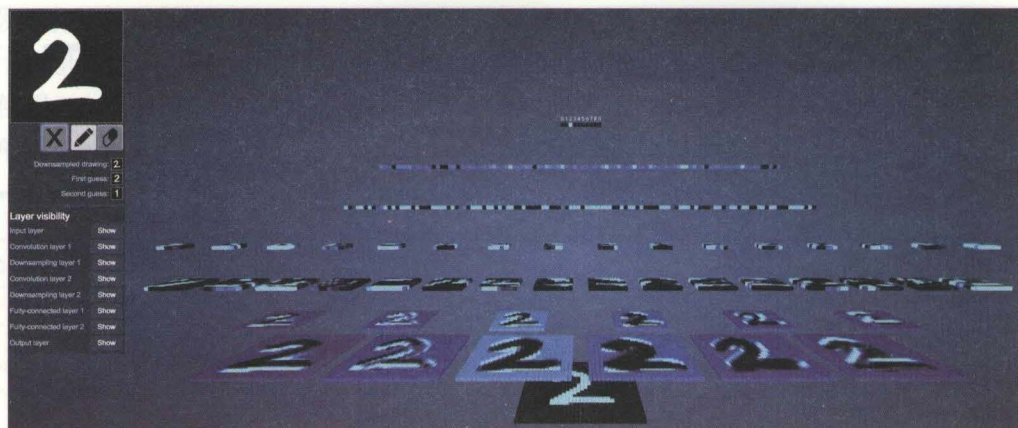


图 1-31 MNIST 识别的成功例子

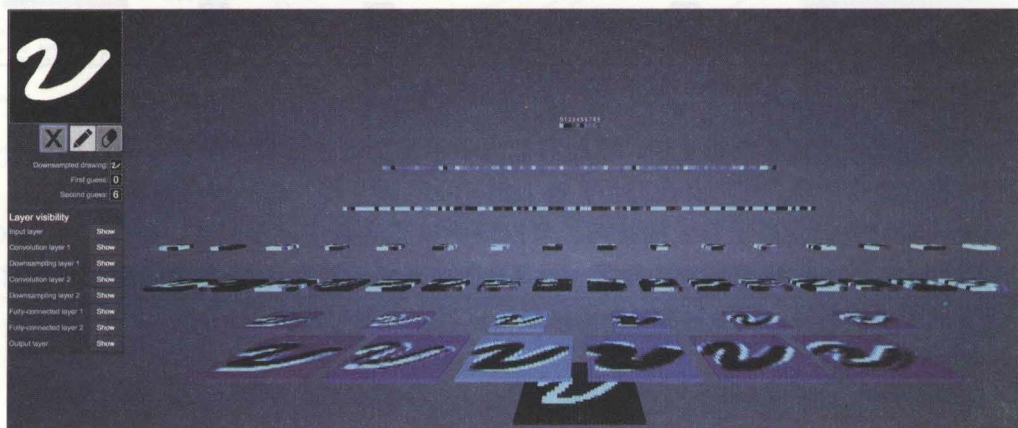


图 1-32 MNIST 识别的失败例子

- ❑ 改进网络架构。
- ❑ 主动生成各种变形数字用于网络训练，让网络提前见识各种变形，提高网络的适应力。这就是后文会介绍的数据增强（data augmentation）方法，是数据威力的体现。

1.4.3 策略网络实例

如本书前言所述，读者在阅读本书后就能训练出自己的策略网络。读者可访问 <https://withablink.coding.me/goPolicyNet/> 先看效果，其中是笔者训练的策略网络，如图 1-33 所示。

由于这个网页需要下载一个较大的网络数据文件，装载需要一段时间，读者如果看不到页面，可刷新试试。

然后 Google 将数据用于训练采用 RNN 架构的 Sketch-RNN 网络。读者可在 https://magenta.tensorflow.org/assets/sketch_rnn_demo/index.html 看到在线演示：选择一个绘画主题，然后画上几笔，电脑会自动将其归为最合适的部位，一笔笔将画补充完整，并且可给出多种答案，如图 1-35 所示。电脑也可直接从零开始生成符合主题的简笔画。

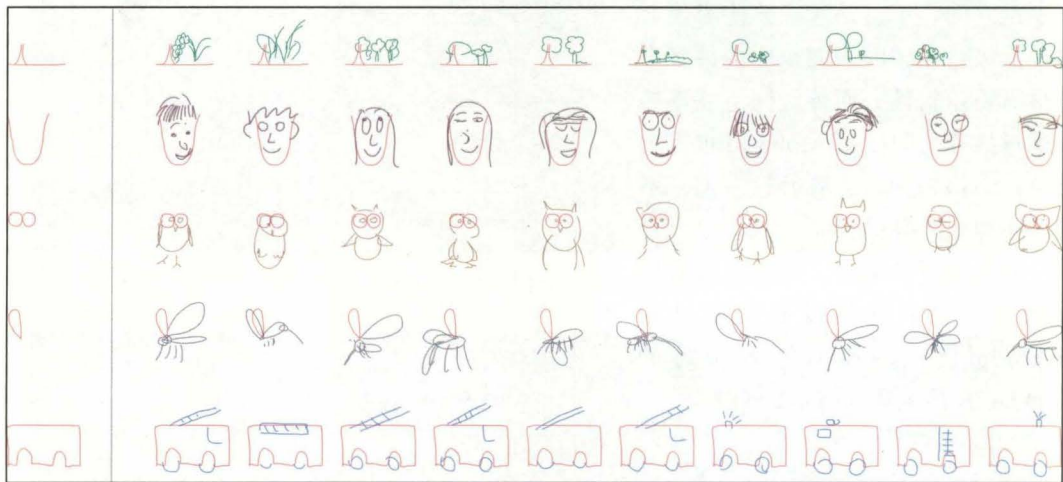


图 1-35 电脑自动生成的简笔画

1.4.5 用 GAN 生成动漫头像

通过使用 GAN，电脑可生成更复杂的图像。

例如，在使用 3 万多张带标签的动漫头像训练后，在 <http://make.girls.moe> 上可根据要求生成拟真度很高的头像，如图 1-36 所示。

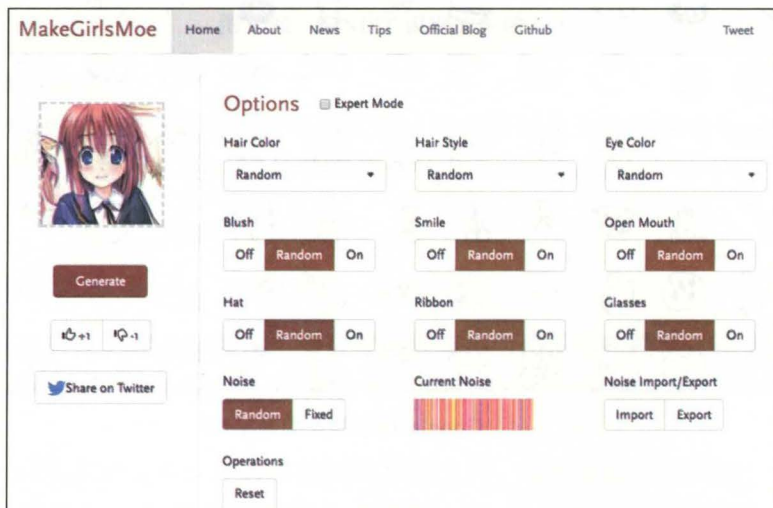


图 1-36 MakeGirlsMoe 的界面

生成的头像效果如图 1-37 所示。读者可在本书第 8 章找到更多关于 GAN 的精彩实例。

1.5 深度神经网络的基本特点

1.5.1 两大助力：算力、数据

在前文我们看过了许多例子，让我们再谈谈理论。虽然深度神经网络的兴起是在近年，但神经网络的思想早在 20 世纪 40 年代就已出现，并在 20 世纪 80 年代就已成型，我们将在后文看到这段历史。

笔者还记得十余年前读到 N.J.Nilsson 的著作《人工智能》，首次了解到神经网络，很有眼前一亮的感觉，但当时也感到神经网络与常规的函数拟合方法相似，当年并没有多少人能预计到神经网络会取得今天的成就。



图 1-37 由 GAN 生成的动漫头像

确实，神经网络的发展曾经历过多年的低谷。那么，它为何在今天焕发出如此的生机？这是因为现在的神经网络规模更大，深度更深，神经元更多，配合更先进的网络架构、更充足的训练数据、更快速的训练硬件，因此效果更好。

□ 更强的算力，让我们可训练更深、更广的网络，实验出更佳的网络架构。

□ 更深、更广的网络，配合更多的数据，就能自动得到更好的效果。

这一切与电脑硬件和互联网的发展息息相关。互联网带来了几乎无穷无尽的数据。研究人员发现，许多复杂的问题，在数据和计算能力足够后，会自动土崩瓦解。

正如著名物理学家 Philip Warren Anderson 所言，“More is different”，量变会引起质变。作为大规模的复杂系统，深度神经网络具有超乎想象的威力，如图 1-38 所示。

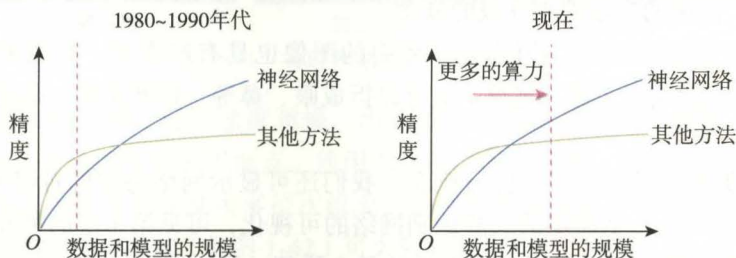


图 1-38 数据与模型的规模，对深度神经网络的促进

深度神经网络对于数据的需求是惊人的。研究人员在《Revisiting Unreasonable Effectiveness

of Data in Deep Learning Era》^①中发现，采用3亿张训练图像进行预训练后，图像分类网络的效果可得到明显提升。而且看上去3亿张训练图像仍没有触及深度神经网络的极限，依旧会发现训练图像越多效果越好。

因此，我们今天所看到的深度神经网络，还拥有着巨大的潜力。正如DeepMind的研究人员发现，AlphaGo Zero的策略网络和价值网络，会随着训练一直提升，棋力的增强宛如没有尽头，虽然它一定存在极限，但很难碰到它的极限。

1.5.2 从特征工程，到逐层抽象

早期的图像分类程序需要人工精心构造许多特征（feature），这称为特征工程。例如，对于识别猫和狗的问题，需要教会电脑如何看出什么是猫毛，猫毛和狗毛的区别是什么；什么是猫眼，猫眼和狗眼的区别是什么……这无疑是个困难且低效率的过程。

这与早期围棋程序所面对的困境相同，当时编程者试图慢慢教会电脑一个个围棋概念。我们不能依靠这种人工的方法，必须找到一个自动化的解决方案。深度神经网络正是答案，如图1-39所示。

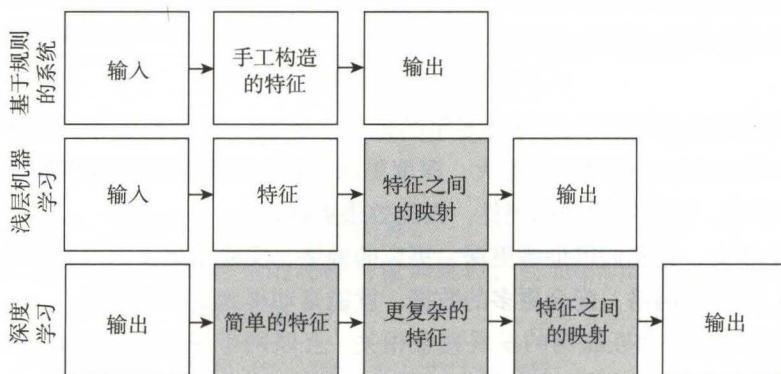


图 1-39 由深度学习自动发现特征的结构

深度神经网络不但可自动发现特征，而且可自动建立逐层的抽象结构：网络的低层处理低级的概念，网络的高层处理高级的概念。

这种逐层的抽象很有效，因为自然界中的图像也具有层次性。以猫的图像为例（见图1-40），猫可以拆成头、身、尾等，头可以拆成眼、鼻等，以此类推，最终可以拆成局部的像素特征。

为证明深度神经网络确实会拆分概念，我们还可显示网络每层的神经元在识别什么。例如，图1-41所示是一个简单的人脸识别网络的可视化，可见第1层的神经元可识别简单的线段边界，第2层的神经元可识别人脸的各个局部，第3层的神经元就可识别相当完整

① 地址为 <https://arxiv.org/abs/1707.02968>。

的人脸。

围棋的情况与此类似，其中高层次概念可拆分为低层次概念，低层次概念可组合为高层次概念。简化的例子是：

- 人在选点时会考虑双方棋子的“厚薄”，这是个高级概念，可认为是棋块的安定性与棋形的结合。
- 如果在网络的某一层，部分神经元能给出安定性的情况，部分神经元能给出棋形的情况，那么，再往上一层就可通过考虑这两种情况，得出类似厚薄的概念。
- 继续往上一层，就可再运用之前得出的厚薄情况，进行进一步的决策。

我们还可举例说明深度神经网络具有有效性的另一原因（请注意，下面是很粗略的看法，但思路是正确的）：

- 如果将 200 个神经元拆分为 2 层，每层 100 个神经元，可产生 $100 \times 100 = 10\,000$ 种组合，比原来的 200 个神经元的表述能力更强。
- 如果拆分为 4 层，每层 50 个神经元，组合的数目就更高达 $50 \times 50 \times 50 \times 50 = 6250000$ 种，表述能力更强。

因此，对于许多复杂问题，网络越深，效果越好。与李世乭对弈的 AlphaGo 使用了十几层的网络，而 AlphaGo Zero 使用了近百层的网络，棋力明显更高。如果使用成百上千层的网络，效果会更好。

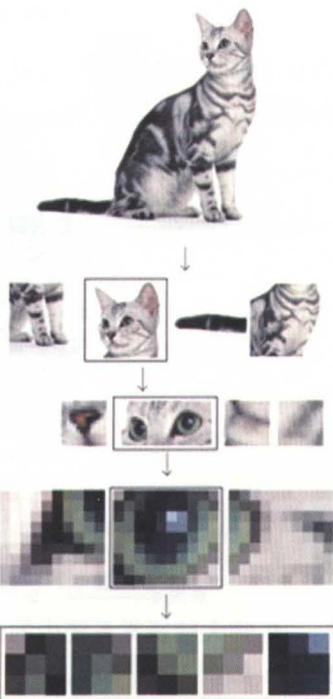


图 1-40 将猫的图像拆分为特征的层次结构



图 1-41 神经网络每层的识别目标

不过，越复杂的网络，运行速度越慢。因此对于下围棋这种对于速度有要求的情况，网络架构需找到一个恰到好处的平衡点。使用 100 层以内的网络，是目前适宜的选择。

生物大脑的视觉系统同样是由多层次组成，也许视觉系统是在进化中形成了这种层次性。例如人类大脑的视觉系统（见图 1-42）可大致划分为五层，从 V1 到 V5（MT）。V1 负责识别简单的概念，如不同角度的线段边界、某些颜色信息等。网络的层次越往后，就越能识别复杂的概念。

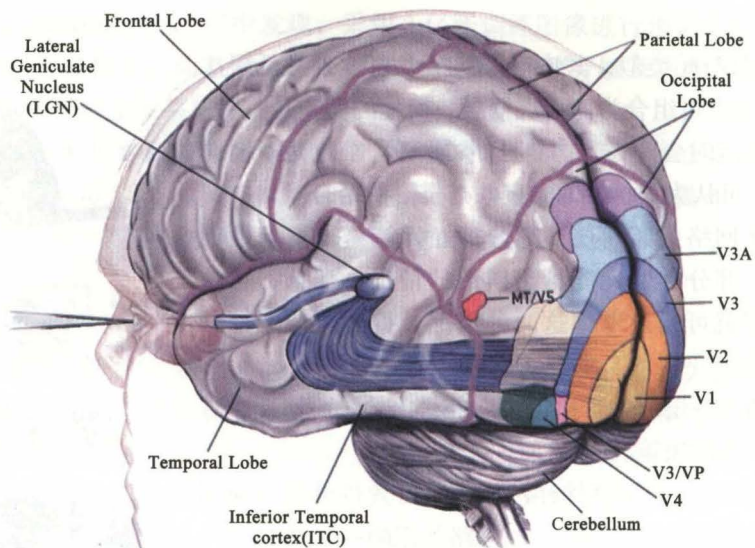


图 1-42 人类的视觉系统

生物大脑视觉系统中的连接非常复杂和密集。猕猴的视觉系统 (Felleman & Van Essen, 1991) 如图 1-43 所示。

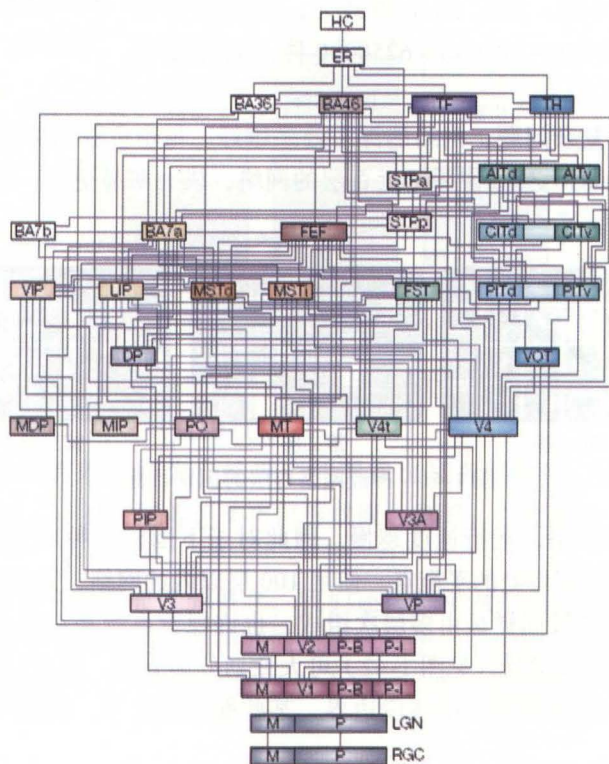


图 1-43 猕猴的视觉系统详图

1.5.3 神经网络学会的是什么

我们在上一节看到，深度神经网络的层次结构与人类大脑类似。深度神经网络学会的规律，如图 1-44 所示，是否和人类脑中的规律一样？例如，AlphaGo 的策略网络，真的学会的是围棋的规律吗？



图 1-44 电脑和人的学习过程

在此，可将规律分为逻辑规律与统计规律：

- 逻辑规律，就像“万有引力”“猫通常有 4 只腿”“星期五之后是星期六”“围棋对杀时长气杀短气，有眼杀无眼”，是确实的真理。
- 统计规律，就像“天上有乌云就可能要下雨”“看到黄白相间的图案就可能是看到了橘猫”“如果全球股市涨，中国股市也可能涨”“围棋中出现愚形三角往往是坏棋”，这更像是某种经验，不一定总是正确。

一直以来，我们认为人的特长是会总结和使用逻辑规律。这确实是人的特长，因为目前的 AI 仍然难以有效地总结和使用逻辑规律。

但，人在很多时候靠感觉就可以判断和处理问题，不需要去思考逻辑规律。例如我们如何判断一幅图像中的动物是猫还是狗？我们可以说出很多逻辑规律，但在我们一眼看出来的时候，我们并不会去明确考虑这些逻辑规律（所以有时我们一眼也会看错）。

这正如职业棋手在下快棋时，瞬间就能有第一感（虽然有时也会看错）。此时我们大脑的运作更接近于参照统计规律。

如果我们仔细分析深度神经网络的内部结构，会发现它实际是学会了各种复杂的统计规律，更接近人的直觉经验（而非逻辑思维）。我们将在后文看到，策略网络所学会的是棋盘上各种棋形的统计规律（类似于各种棋形出现的概率），以及各种棋形的相互组合和影响的统计规律（类似于条件概率）。

因此深度神经网络有时也会犯低级错误。在这个意义上，深度神经网络和我们大脑的

生物神经网络的直觉有某种深层次的相似之处。

另外，对于围棋这种具有很强逻辑性的问题，人在静心思考的时候，会计算许多事情，考虑很多逻辑规律，而不是只靠感觉。因此，单纯靠策略网络，电脑的棋力并无法超越人类。所以 AlphaGo 还会使用多种其他组件，包括逻辑性很强的蒙特卡洛树搜索，相互配合，就有能力修正直觉的错误，趋于完美。在电脑同时拥有了直觉和逻辑后，棋力就拥有了巨大的上升空间。

需要指出，统计规律的威力可超越我们的想象。例如，在机器翻译方面，目前最新的深度神经网络[⊖]已能做到这样的事情：只要给网络看充足的 A 语言的资料，和充足的 B 语言的资料，无须给出任何 A 语言和 B 语言之间翻译的例句，网络就能自动找到在 A 语言和 B 语言之间翻译的方法。

这就像将一套中文版的《红楼梦》和一套法文版的《追忆似水年华》给一位不识字的人，然后他就能找到中文和法文之间的翻译方法，自己编出一本《汉法词典》，如图 1-45 所示。这听上去非常不可思议，然而深度神经网络真的就做到了这样不可思议的事情。

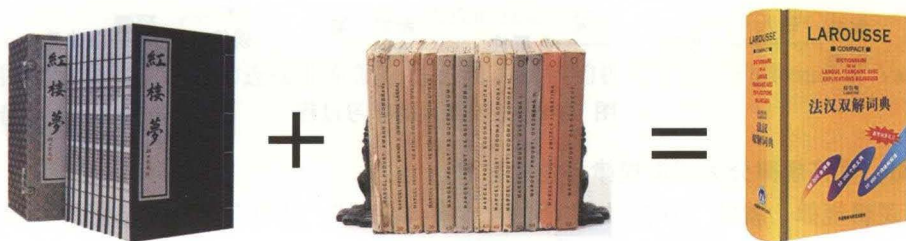


图 1-45 深度神经网络足以全自动学会翻译

从统计规律的角度可以理解这个过程。因为，可仔细分析每个汉字和法文字母的统计规律、汉字组合成词语的规律、字母组合成词语的规律、词语之间的组合规律，慢慢就可看出两边的词语的对应关系、语法的对应关系，最终实现翻译。这个过程无疑非常艰巨，但 AI 不会害怕，它可以慢慢把所有线索理清楚。AI 在此的“学习能力”已是人类所难以企及的。

我们能感到庆幸的是，AI 在这个过程中仍然会犯错。目前的 AI，实际仍然没有真正理解人类语言，因为它仍然缺乏人类的逻辑思维能力，它只能“像理解语言”。同样，虽然 AI 做图像分类的准确率已胜过人类，但它其实也只是“像理解图像”。本书的第 9 章将继续这里的讨论。

1.6 人工智能与神经网络的历史

在本章的最后，让我们回顾 AI 和神经网络的历史。

⊖ 地址为 <https://arxiv.org/abs/1710.11041> 和 <https://arxiv.org/abs/1711.00043>。

1.6.1 人工智能的两大学派：逻辑与统计

上节提到了逻辑规律和统计规律。在AI的历史上，也曾有逻辑规律和统计规律之争。具体而言，人工智能有两大主流学派：

- 符号主义 (symbolism)，又称为计算主义 (computationalism) 或逻辑主义 (logicism)，它认为智能需要通过精确的逻辑推理计算实现。这是传统的人工智能方法。从前的例子是专家系统和知识库，现在的例子是知识图谱，例如企业关系图，如图 1-46 所示。

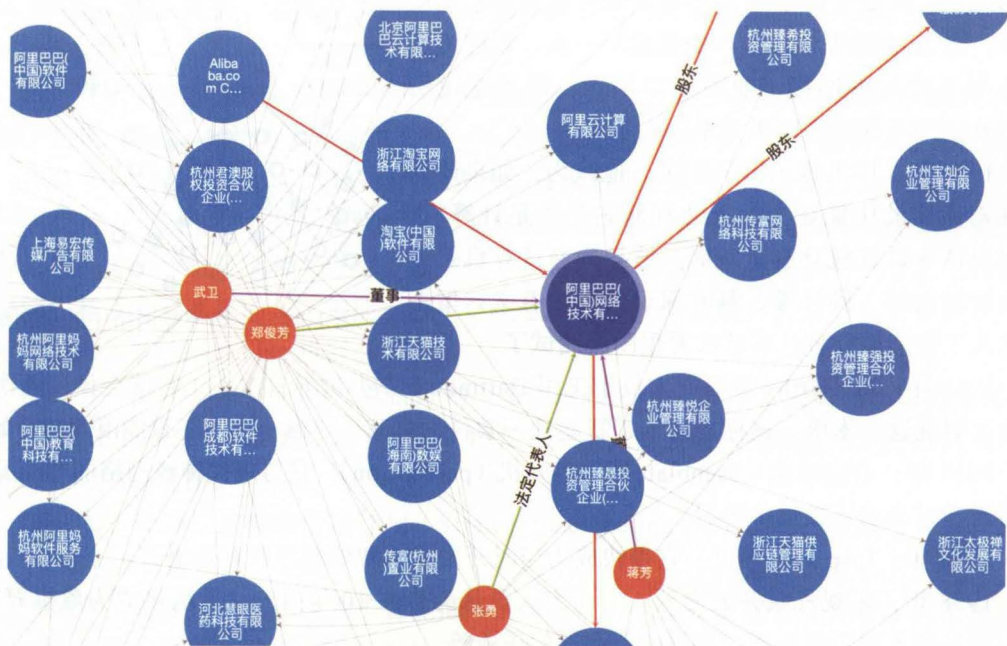


图 1-46 知识图谱的例子：企业关系图

- 统计主义，它更注重从数据和经验中学习统计规律。目前绝大多数机器学习方法都属于统计主义，因为我们目前仍然缺乏让电脑自动提炼和运用逻辑规律的有效方法。例如，连接主义 (connectionism)，它认为智能需要模拟生物大脑的神经元结构，当大量简单的单元被连接在一起时，就可以实现智能行为。这就是人工神经网络及其训练方法。

这两大学派各有所长，都曾经历兴衰。人工智能也曾经历两次浪潮与两次低谷，目前我们正位于第三次浪潮。这一切与计算机硬件的发展有着密不可分的联系。

1.6.2 人工智能与神经网络的现代编年史

在两千多年前，从春秋战国时期《列子》中描述的偃师造人，到古希腊传说中的青铜机器人 Talos，人类就曾梦想创造出与人类具有相同行为模式的智能机械。而 17 世纪

Leibnitz 的数理逻辑与 Pascal 的机械加法机，可谓人工智能算法和硬件的先驱。

我们将人工智能与神经网络的现代编年史总结如下：

1936 年：数学家图灵提出图灵机，奠定了计算理论的基础。

1943 年：神经科学家 McCulloch 与逻辑学家 Pitts 提出简化的神经元模型，但当时的神经元参数需要人工预先设定。

1946 年：数学家冯·诺依曼提出计算机的冯·诺依曼架构。此架构沿用至今。

1949 年：心理学家 Hebb 提出了生物神经网络的训练机制。即如果神经元 A 与 B 常常在相近的时刻被激活，且 A 的激活总是比 B 的激活早一点，就应该增强 A 与 B 之间的连接，如图 1-47 所示，因为这说明 A 与 B 之间可能存在因果关系。

1950 年：图灵提出图灵测试。他认为，如果计算机在对话实验中能让人类无法判断它是否是计算机，就足以认为计算机具有了智能。这是判断计算机是否具备智能的第 1 种标准，具有深远的历史意义，但今天的人工智能研究者已不那么看重图灵测试了。

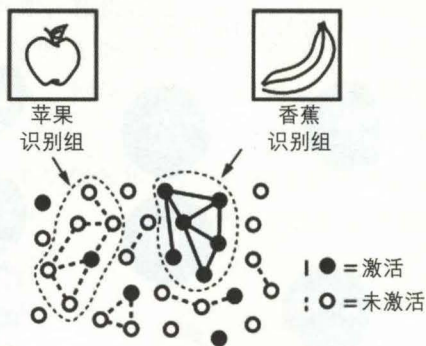


图 1-47 生物神经网络中的识别组

1956 年：McCarthy 等研究人员于美国 Dartmouth 举办世界首次人工智能会议，首次使用人工智能这一术语，这标志着人工智能这一学科的奠基，与第一次人工智能浪潮的开始。

1957 年：心理学家 Rosenblatt 提出感知机 (perceptron)，它与现代神经网络的神经元已经接近，可自动从数据中学习。

1958 年：Lisp 语言面世，它很快成为当时开发人工智能程序的首选语言。

1958 年：乐观气氛开始蔓延。有研究人员认为，在 10 年内，计算机将成为象棋冠军、发现和证明重要数学定理、谱写专业水准的乐曲等。

1963 年：计算机科学家 Vapnik 提出了支持向量机 (Support Vector Machine, SVM)，它的基本原理是使用核函数变换数据，再用决策平面划分数据，如图 1-48 所示，我们稍后将再次看到它。

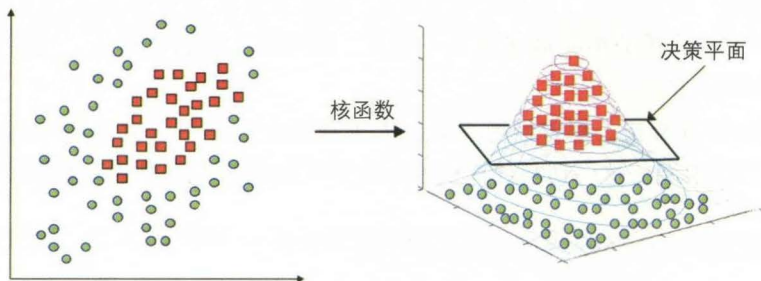


图 1-48 支持向量机的基本原理

1968年：认知科学家 Minsky 指出，当时的原始神经网络无法解决许多简单的问题，包括著名的 XOR 问题。这导致了神经网络的第一次衰落。

1972年：Prolog 语言面世，这是一种基于逻辑的编程语言，是符号主义的代表。

1973年：数学家 Lighthill 向英国科学委员会提交了著名的报告，认为当时的人工智能在多个领域不可能达到所宣称的目标。这确实属实。英国将投向人工智能的研究经费大幅削减，美国也随后跟随，第一次人工智能的冬天降临了。

1974年：计算机科学家 Werbos 在博士论文中提出可以使用反向传播（Back Propagation, BP）算法，实现神经网络的训练。但由于当时的电脑能力有限，这并未引起任何波澜。

1980年：哲学家 Searle 提出中文房间（Chinese room）实验。他认为，如果计算机只是按照程序行事，那么，即使它表现得和人类一样，例如通过图灵测试，也不能被认为具有智能。

1981年：IBM PC 面世。电脑硬件的发展，逐渐带来了人工智能的第二次崛起。

1981年：进入 20 世纪 80 年代后，人工智能全面复苏，基于 Lisp 和 Prolog 的专家系统日益流行，大学纷纷开设相关课程。据估计，当时的世界 500 强公司中有三分之二在研究如何使用基于专家系统的人工智能。数十亿美金被投往人工智能，大量的人工智能专用硬件开始出现，当时称为 Lisp 机器。

1982年：日本启动第五代计算机计划，希望通过 Prolog 等语言突破冯·诺依曼架构，在 10 年内让电脑在语言识别、语音识别、图像识别上达到接近人类的水平，并取代美国成为人工智能的领导者。

1983年：美国启动战略计算计划（Strategic Computing Initiative），这是与日本针锋相对的 10 年人工智能计划。

1984年：曾经历过第一次人工智能低谷的 Minsky 和 Schank 警告业界对于人工智能的热情已经失控，并提出了“人工智能冬天”概念。人工智能冬天有四个阶段，即研究者开始感到悲观 → 悲观情绪被媒体渲染 → 研究经费被削减 → 研究彻底进入低谷。

1986年：计算机科学家 Hinton 和心理学家 Rumelhart 表示，通过运用反向传播，可以让神经网络的中间层学会有价值的数据表示（representation）。对于神经网络的研究在学术界逐渐复苏。但当时的神经网络的训练速度过慢，且反向传播容易陷入局部极值，因此许多研究人员并不看好神经网络。研究人员直到多年之后才发现，这实际是因为当时的神经网络规模仍然太小。

1987年：伴随着 1987 年股灾，Lisp 机器市场崩盘，第二次人工智能的冬天开始降临。

1988年：美国的战略计算计划放弃向人工智能的投入，转向超级计算机研究。

1992年：因最终未能突破技术难题，日本的第五代计算机计划在耗费 500 亿日元后宣告失败并转型。

我们以例子说明专家系统在技术上的困难。著名的项目是 Cyc 知识库[⊖]，举例说明其中

⊖ 地址为 <http://www.cyc.com/>。

的内容：

```
克林顿是美国总统之一：
(#$isa #$BillClinton #$UnitedStatesPresident)
树木都是植物：
(#$genls #$Tree-ThePlant #$Plant)
巴黎是法国的首都：
(#$capitalCity 1#$France #$Paris)
```

然而 Cyc 不能理解一个名为 Fred 的人用电动剃须刀剃须的故事，因为 Cyc 知道人的构成不包括电子零件，但 Cyc 认为“正在剃须的 Fred”包含有电动剃须刀的电子零件，因此 Cyc 无法确定“正在剃须的 Fred”是否仍然是一个人。

专家系统的另一个致命缺陷是，电脑不能自动发现规律，仍然需要大量人工维护。因为专家系统使用的是逻辑规律。电脑迄今仍然不擅长发现和使用逻辑规律。

1995 年：虽然人工智能在业界的泡沫破灭，但研究仍在学术界继续。经过 30 多年不断改进，支持向量机（SVM）终于成为机器学习的主流。它的性能在当时最佳，也具有有良好的数学性质，直至 2012 年被深度神经网络取代为止，一直是机器学习中最热门的算法。

由于 SVM 的优势，极少有研究人员会再去关注神经网络，神经网络进入第二次冬天。

1998 年：神经网络研究在暗中不断发展。以 Lecun 为首的研究人员提出了 LeNet-5 网络结构，在 MNIST 数据集上实现了当时最好的识别准确度，胜过多种其他机器学习方法。但神经网络仍然处于黑暗时期，如果研究人员的论文中出现了神经网络一词，就可能会因此被拒稿。

2006 年：以 Hinton 为首的研究人员提出了深度信念网络（这是一种需逐层训练的深度神经网络，也是首个容易训练的深度网络），并将深度神经网络包装为“深度学习”进行推广，因为神经网络一词在当时已经“声名狼藉”。

在神经网络的黑暗时期，Hinton、Lecun 与另一位研究者 Bengio 始终坚持研究和推广神经网络，这三位今天常被称为是深度学习的“三巨头”。

小插曲：虽然无关学术，但命名与包装对于人工智能中技术的推广往往确实具有重要性。例如，SVM 的全名 Support Vector Machine 中使用了“机器”（machine）一词，并不是因为它真的是一台机器，而只是为了让它在面对业界时显得更专业（让用户感觉好像获得了一台机器）。又例如专家系统在被业界抛弃后，在近年被包装为知识图谱，听上去更时髦，重新获得研究资金的支持。

2012 年：以 Hinton 为首的研究人员在 ImageNet 2012 图像识别大赛上夺冠，他们的深度卷积网络 AlexNet 的识别准确率远远胜过第二名。深度学习的热潮至此终于拉开大幕。

深度神经网络崛起最重要的原因是互联网和计算机硬件的发展。互联网提供了取之不尽的数据，而 GPU 的发展带来了训练速度的数量级提高，因此可使用更为复杂的网络。这是一个量变引起质变的过程。

例如，在过去，研究人员一直不相信像反向传播 BP 这样简单的算法就可以成功训练深度网络。但我们今天训练深度网络所使用的正是最简单的反向传播，确实出人意料。

2013 年：据 Bostrom 对几百位人工智能专家的调查显示，研究人员对于“能够在大多数职业上达到典型人类的水平”的强人工智能的实现年份的中位数估计是 2040 年。

2016 年：在 AlphaGo 战胜李世乭后，Google 公司发布了深度学习的专用硬件 TPU。

2017 年：从图像到语音、语言，深度学习几乎统治了所有机器学习领域，而 SVM 等传统方法已少有人问津（可能已出现反向歧视）。知名公司都在基于深度学习的人工智能上投入重金。初创公司都会为自己贴上人工智能的概念。深度学习专用硬件公司开始涌现。

这令人回想起 30 年前的历史，目前也有业界人士认为现在的 AI 存在一定的泡沫。但是，由于数据和算力的助力，这次的人工智能确实比从前显著更强。

由于数据和算力的大幅提升，人工智能的统计与逻辑流派在今天都展现出欣欣向荣的景象，这是历史上从未有过的繁荣局面。或许最终的答案就是结合深度网络、逻辑推理计算、知识库、深度学习、强化学习、预测学习，实现具有自我进化能力的人工智能，如图 1-49 所示。

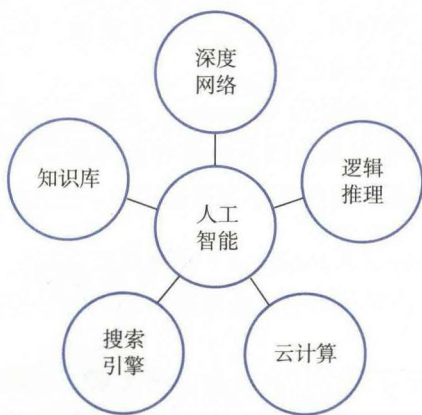


图 1-49 人工智能的一种组合结构

深度卷积网络：第一课

经典的神经网络，是多层感知机（MLP），又称全连接（fully connected）神经网络，因为它的层与层之间的神经元相互完全连接。

神经网络的基本运作单位是神经元（neuron），类似于生物大脑的神经元，如图 2-1 所示。注意电脑和生物神经网络有区别，因为电脑神经网络采用浮点数作为神经元之间的通信手段；而生物神经网络采用脉冲电信号作为通信手段，包含时间因素，更为微妙。

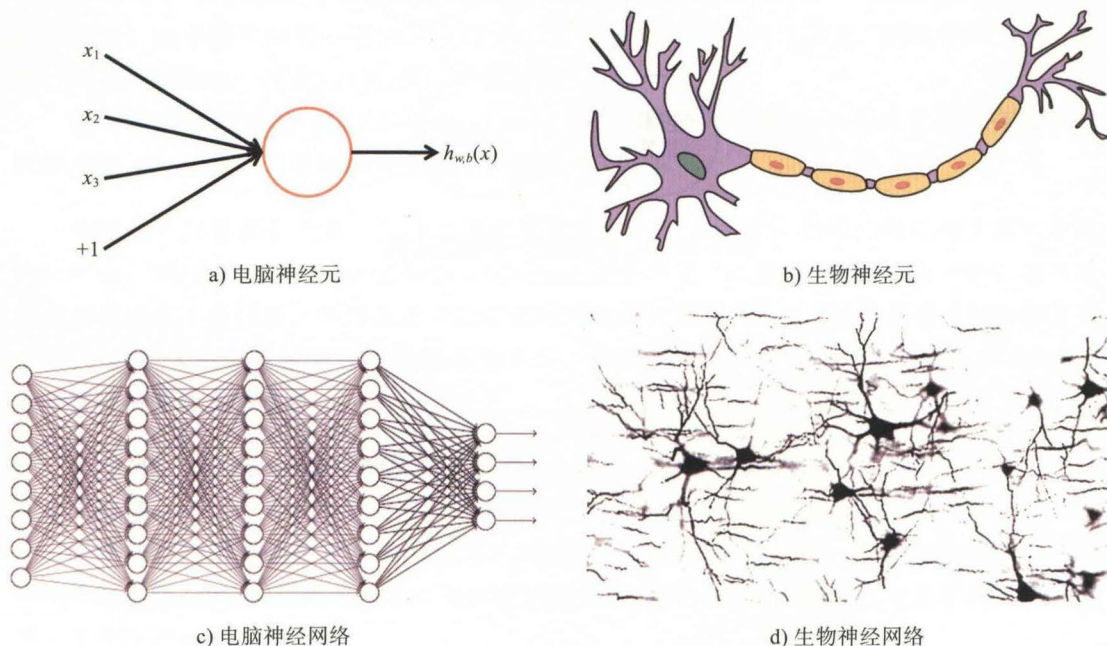


图 2-1 电脑与生物神经网络的对比

如前所述，我们可将神经网络看作一个复杂的函数，它有能力拟合复杂的数据。神经网络相对于其他拟合方法，有几大优势：

- ❑ 它可拥有极大的参数量（常用的模型可有几千万个参数），可拟合极其复杂的数据。用专业术语说，它的容量（capacity）很大，可避免欠拟合（under-fit）。
- ❑ 它的运行速度很快，容易并行化，在 GPU 上尤其快，在专用硬件（如 Google 研发的 TPU）上更快。换言之，我们可快速运行和训练极大规模的神经网络。
- ❑ 它的训练过程（即拟合过程）有快速而有效的方法：反向传播（back-propagation, BP）、随机梯度下降（stochastic gradient descent, SGD）。
- ❑ 传统观点是，如果参数量太大，模型就容易出现过拟合，即只是死记硬背，没有真正发掘数据背后的规律。但目前的神经网络可在诸多问题上成功避免过拟合，效果超越其他模型，近乎于全自动学会解决问题的方法，令人赞叹。

2.1 神经元：运作和训练

我们先详细讲述单个神经元的运作和训练，这会涉及许多关键的概念。

这里会用到不少数学知识，如果读者不熟悉数学，可先跳过看不明白的部分。因为目前的深度学习框架已很完善，即使用户不了解数学，不了解神经网络的工作原理，仍然可以训练，就像即使不了解汽车的原理仍然可以开车（只是可能会开得没那么好）。

2.1.1 运作：从实例说明

直观地说，每个神经元是在做加权平均（weighted average），类似于用线性模型决策^①。例如，某个最简化的预测明天的股市上证指数开盘价格的模型（注意这是假想模型，请勿根据此模型投资）：

- ❑ 明天的股市开盘价格应该和今天的股市收盘价格差不多。例如，如果昨天收市是 3212.45 点，今天开盘也差不多会是 3212.45 点。因此这个因素的权重（weight）为 1。
- ❑ 今天晚上的美国股市的变动情况，会对其有一定影响。例如，如果美国股市的 S&P 指数每变动 X 点，会对上证指数有 $X \cdot 0.5$ 点的影响，那么这个因素的权重为 0.5。^②
- ❑ 股市还有长期缓缓向上的趋势（虽然中国股市可能在此略有欠缺）。根据长期趋势，假设每天股市平均会上升 0.1 点。这可称为偏置（bias）。

为此，可按图 2-2 所示构建模型，图中用连接上的数字代表连接的权重。

这就是 1 个神经元。它有 2 个输入（今天收盘价格、今晚美股走势），1 个输出（明天开盘价格），还有 3 个参数（2 个权重，1 个偏置）。

举例，如果今天上证指数收盘价格是 3212.45 点，今晚美国 S&P 指数下跌 1.68 点，那么

① 在实际中，我们还会加入非线性激活函数，稍后的 2.3 节就会讨论。

② 这是假设，实际不一定是 0.5。

神经元对于明天上证指数开盘价格的预测是：

$$3212.45 \cdot 1 + (-1.68) \cdot 0.5 + 0.1 = 3211.71$$

上述模型中的参数（权重和偏置）是手工写入的。而在实际的神经网络中，我们会通过训练自动从数据找到神经元的参数，稍后会讲述训练过程。

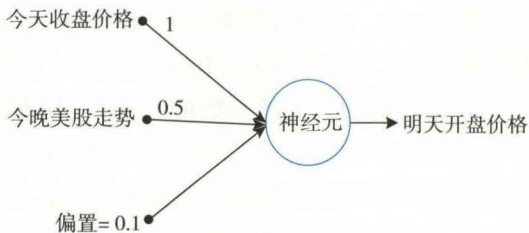


图 2-2 神经元的实例

一般而言，每个神经元有多个输入 x_i

（我们可按需求设计输入的个数），一个输出 OUT（如果需要多个输出，可使用多个神经元），以及两类参数：

- 权重，可写为 w_i 。神经元的每一个输入 x_i 都会乘以一个权重 w_i ，可认为它标志着每一个输入的重要性。权重是实数，可以取正值和负值。
- 偏置，可写为 b 。神经元的输出常常会加上一个偏置 b ，类似于线性模型的常数项。偏置也是实数，可以取正值和负值。

注意，神经元的设计可以很灵活。我们有时会省略偏置，有时会在多个神经元间共享权重。继续举例。图 2-3 所示的神经元有 3 个输入： x_1, x_2, x_3 。

这个神经元的最终输出公式是：

$$\text{OUT} = x_1 w_1 + x_2 w_2 + x_3 w_3 + b$$

总而言之，公式是：

$$\text{OUT} = \left(\sum_i x_i \cdot w_i \right) + b$$

此外，有的文献会把偏置看成是一个“固定为 1”的输入的权重，如图 2-4 所示。

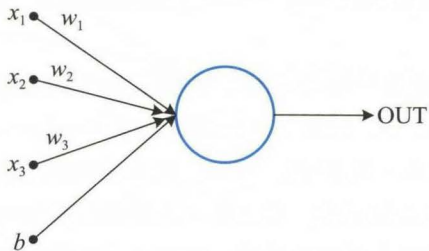


图 2-3 带有 3 个输入的神经元

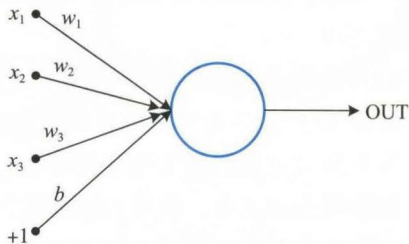


图 2-4 神经元的一种等价画法

这是因为可将之前的公式写成：

$$\text{OUT} = \left(\sum_i x_i \cdot w_i \right) + 1 \cdot b$$

那么 b 就像是一个“固定为 1”的输入权重。

2.1.2 训练：梯度下降的思想

神经元和神经网络的训练思想一致，因此我们在此直接讲述神经网络的训练思想。首

先，神经网络的基本运作过程如图 2-5 所示。

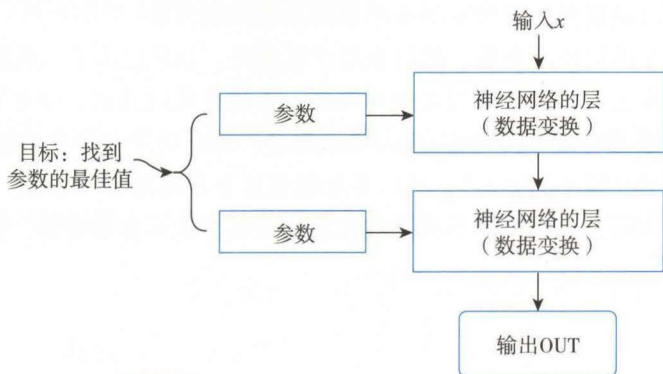


图 2-5 神经网络的基本运作

神经网络的每层会对数据进行变换，变换的方法由参数决定。神经网络的训练过程是逐步调整神经网络每层的参数，让神经网络的输出 OUT 趋近于期望输出。

为此，需要衡量什么是“趋近”，定义损失函数（loss function），代表 OUT 和期望输出之间的误差，然后训练的目标是最小化损失函数的值 LOSS。

例如，均方差（mean squared error, MSE）损失（又称 L2 损失）为：

$$\text{LOSS} = (\text{OUT} - \text{期望输出})^2$$

由公式可见，此 LOSS 在 OUT 等于期望输出时，才能取得最小值，且最小值为 0。

在定义损失函数后，神经网络的训练过程如图 2-6 所示。

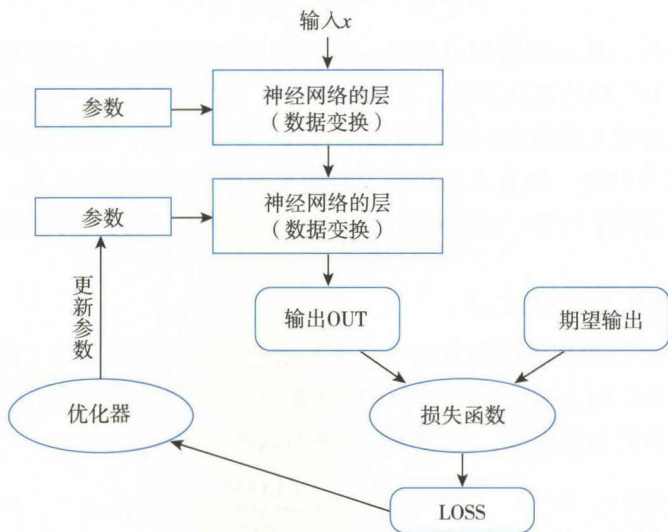


图 2-6 神经网络的训练过程

将数据 X 输入神经网络，得到输出 OUT。将 OUT 与期望输出 Y 比较，得到 LOSS。然后

优化器 (optimizer) 会根据 LOSS 和神经网络的情况, 更新神经网络的参数, 使得 LOSS 缩小。

这里优化器的目的是找到能让 LOSS 缩小的参数调整方法。

最原始的方法是随机微调参数, 然后重新计算损失。如果进步了, 就保留这个微调; 如果退步了, 就还原参数。这个方法无疑效率很低, 不过也可以改进, 学术界对此也有研究。

更好的训练方法是梯度下降 (gradient descent)。可通过求偏导数算出往哪个方向调整参数是“最能使 LOSS 缩小的方向”, 然后将参数朝这个方向移动一小步。

这个过程就像下山, 一直朝着下山的坡度最陡峭的方向一步步行走, 如图 2-7 所示。

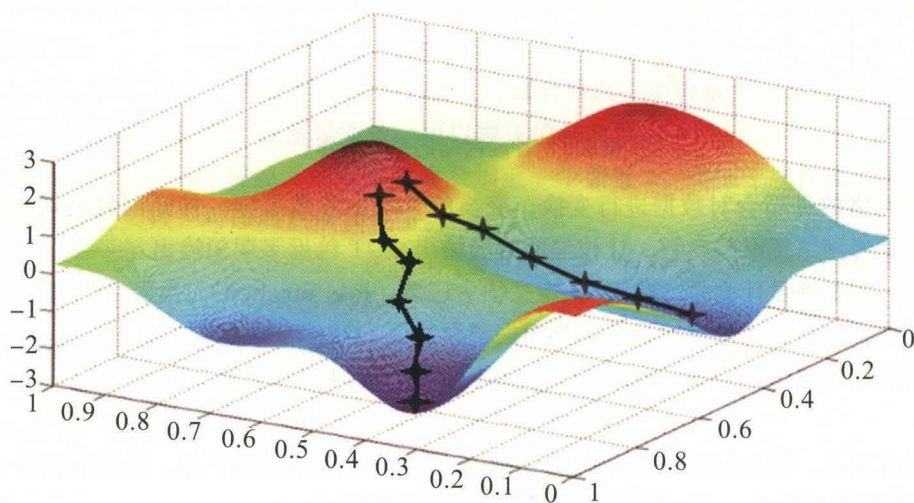


图 2-7 梯度下降

从图中也可看到, 从不同的起点开始, 可以到达不同的终点, 最终的 LOSS 也很可能不同。不过, 对于大规模的神经网络, 这实际影响不大, 后文会进一步探讨。

这个训练过程也让人想起我们练习投篮的过程: 不断尝试, 边试边调整, 最后投准了。但严格说这个比喻有问题, 因为人脑在学习时似乎并没有求导数的过程, 生物神经网络的学习机制和电脑神经网络的学习机制有微妙区别, 这是学术界仍在研究的课题。

2.1.3 训练：梯度下降的公式

在此我们看神经元的具体训练方法。这里开始会出现更多的数学。如前所述, 读者可先跳过看不懂的部分, 因为现在的深度学习框架都会自动计算这些事情。

对于神经网络的任何参数 w , 使用梯度下降的训练公式都是:

$$w^{\text{new}} = w - \eta \cdot \frac{\partial \text{LOSS}}{\partial w}$$

其中 η 称为学习速率 (learning rate), 是一个我们在训练前指定的很小的正数 (例如 0.01), 类似于梯度下降的每一步的步长。而 ∂ 是求偏导数, $\frac{\partial \text{LOSS}}{\partial w}$ 称为梯度 (gradientd)。

如果读者回想导数的定义，会看出上述公式可将 w 往最能快速减少 LOSS 的方向移动。令 LOSS^{new} 为 w^{new} 对应的新 LOSS，根据导数的定义，计算得：

$$\text{LOSS}^{\text{new}} \approx \text{LOSS} - \eta \cdot \left(\frac{\partial \text{LOSS}}{\partial w} \right)^2 < \text{LOSS}$$

由于新的 LOSS^{new} 比旧的 LOSS 小，故实现了 LOSS 的下降。如果读者疑惑这个式子是怎么算出来的，稍后有推导过程。

再看神经元的具体梯度下降公式。假设我们使用此前所述的 MSE 损失：

$$\text{LOSS} = (\text{OUT} - \text{期望输出})^2, \quad \frac{\partial \text{LOSS}}{\partial \text{OUT}} = 2(\text{OUT} - \text{期望输出})$$

那么，对于神经元的权重，公式为：

$$\begin{aligned} w_i^{\text{new}} &= w_i - \eta \cdot \frac{\partial \text{LOSS}}{\partial w_i} \\ &= w_i - \eta \cdot \frac{\partial \text{LOSS}}{\partial \text{OUT}} \cdot \frac{\partial \text{OUT}}{\partial w_i} \\ &= w_i - \eta \cdot 2(\text{OUT} - \text{期望输出}) \cdot x_i \end{aligned}$$

对于偏置，公式为：

$$\begin{aligned} b^{\text{new}} &= b - \eta \cdot \frac{\partial \text{LOSS}}{\partial b} \\ &= b - \eta \cdot \frac{\partial \text{LOSS}}{\partial \text{OUT}} \cdot \frac{\partial \text{OUT}}{\partial b} \\ &= b - \eta \cdot 2(\text{OUT} - \text{期望输出}) \end{aligned}$$

最后推导刚才的式子。根据导数的定义，对于任何可导函数 f ，有：

$$f(t - \varepsilon) \approx f(t) - \varepsilon \cdot \frac{\partial f}{\partial t}$$

因此：

$$\text{OUT}(w^{\text{new}}, x) = \text{OUT}\left(w - \eta \cdot \frac{\partial \text{LOSS}}{\partial w}, x\right) \approx \text{OUT}(w, x) - \eta \cdot \frac{\partial \text{LOSS}}{\partial w} \cdot \frac{\partial \text{OUT}}{\partial w}$$

那么：

$$\begin{aligned} \text{LOSS}^{\text{new}} &= \text{LOSS}(\text{OUT}(w^{\text{new}}, x)) \\ &\approx \text{LOSS}\left(\text{OUT}(w, x) - \eta \cdot \frac{\partial \text{LOSS}}{\partial w} \cdot \frac{\partial \text{OUT}}{\partial w}\right) \\ &\approx \text{LOSS}(\text{OUT}(w, x)) - \eta \cdot \frac{\partial \text{LOSS}}{\partial w} \cdot \frac{\partial \text{OUT}}{\partial w} \cdot \frac{\partial \text{LOSS}}{\partial \text{OUT}} \\ &= \text{LOSS}(\text{OUT}(w, x)) - \eta \cdot \left(\frac{\partial \text{LOSS}}{\partial w} \right)^2 \end{aligned}$$

如果读者对于这里的第2行到第3行的推导有困惑，请注意 $\text{LOSS}()$ 是函数，不是乘法，所以需要套用导数的公式。

2.1.4 训练：找大小问题的初次尝试

让我们用实际问题说明神经元的训练过程。假设我们希望找到一个神经元，满足下列要求：

- 输入是 x_1 和 x_2 ，且是两个 0 到 1 之间的数字。
- 输出 OUT 尽量接近这两个数字中较大的那个，即 $\text{MAX}(x_1, x_2)$ 。
- 注意：如果读者熟悉数学，会意识到我们不可能找到单个神经元使 OUT 永远等于 $\text{MAX}(x_1, x_2)$ ，但确实可找到一个能使 OUT 尽量接近 $\text{MAX}(x_1, x_2)$ 的神经元。为此，首先生成训练数据集，其中是随机的训练样本，如表 2-1 所示。

表 2-1 训练样本

样本编号	x_1	x_2	期望输出
1	0.1	0.8	0.8
2	0.5	0.3	0.5
...

令学习速率 η 为 0.1，则训练过程如下：

初始参数： $w_1=0, w_2=0, b=0$ 。注意，虽然这里的初始参数设置为全 0，但对于更复杂的网络，初始参数不能设置为全 0，后文会解释原因。

输入： $x_1=0.1, x_2=0.8$ 。

输出：

$$\text{OUT} = w_1 x_1 + w_2 x_2 + b = 0 \cdot 0.1 + 0 \cdot 0.8 + 0 = 0$$

期望输出=0.8。

损失：

$$\text{LOSS} = (\text{OUT} - \text{期望输出})^2 = (0 - 0.8)^2 = 0.64$$

回顾此前的公式：

$$w_i^{\text{new}} = w_i - \eta \cdot 2(\text{OUT} - \text{期望输出}) \cdot x_i$$

$$b^{\text{new}} = b - \eta \cdot 2(\text{OUT} - \text{期望输出})$$

因此，新权重：

$$w_1^{\text{new}} = w_1 - \eta \cdot 2(\text{OUT} - \text{期望输出}) \cdot x_1 = 0 - 0.1 \cdot 2 \cdot (0 - 0.8) \cdot 0.1 = 0.016$$

$$w_2^{\text{new}} = w_2 - \eta \cdot 2(\text{OUT} - \text{期望输出}) \cdot x_2 = 0 - 0.1 \cdot 2 \cdot (0 - 0.8) \cdot 0.8 = 0.128$$

$$b^{\text{new}} = b - \eta \cdot 2(\text{OUT} - \text{期望输出}) = 0 - 0.1 \cdot 2 \cdot (0 - 0.8) = 0.16$$

继续训练下一组样本：

输入： $x_1=0.5$ ， $x_2=0.3$ 。

输出：

$$\text{OUT} = w_1x_1 + w_2x_2 + b = 0.016 \cdot 0.5 + 0.128 \cdot 0.3 + 0.16 = 0.2064$$

期望输出=0.5。

损失：

$$\text{LOSS} = (\text{OUT} - \text{期望输出})^2 = (0.2064 - 0.5)^2 = 0.0862$$

因此，新权重：

$$w_1^{\text{new}} = w_1 - \eta \cdot 2(\text{OUT} - \text{期望输出}) \cdot x_1 = 0.016 - 0.1 \cdot 2 \cdot (0.2064 - 0.5) \cdot 0.5 = 0.04536$$

$$w_2^{\text{new}} = w_2 - \eta \cdot 2(\text{OUT} - \text{期望输出}) \cdot x_2 = 0.128 - 0.1 \cdot 2 \cdot (0.2064 - 0.5) \cdot 0.3 = 0.145616$$

$$b^{\text{new}} = b - \eta \cdot 2(\text{OUT} - \text{期望输出}) = 0.16 - 0.1 \cdot 2 \cdot (0.2064 - 0.5) = 0.21872$$

.....

训练几百个样本后，会看到参数在 $w_1 = 0.5 = \frac{1}{2}$ ， $w_2 = 0.5 = \frac{1}{2}$ ， $b = 0.16666 \dots = \frac{1}{6}$ ，

附近振荡。

此时可缩小学习速率 η ，例如缩小到 0.02，继续训练，并且可随着训练的情况继续缩小 η

(后文会解释为何需要逐渐缩小学习速率)。最终会收敛到很接近 $w_1 = \frac{1}{2}$ ， $w_2 = \frac{1}{2}$ ， $b = \frac{1}{6}$ 。

此外，如果改变选取样本的顺序，例如先选取 2 号样本，然后再选取 1 号样本，将得到不同的权重。读者可尝试计算。我们希望神经网络尽量少受到训练顺序的影响，因此在每次训练时都会打乱样本，让训练顺序随机化，这有助于最终得到更好的效果，避免过拟合。

最后，我们证明 $w_1 = \frac{1}{2}$ ， $w_2 = \frac{1}{2}$ ， $b = \frac{1}{6}$ 就是理论最优值。为简化证明，这里只考虑

更简单的情况，令 $w_1=w_2=w$ 。

我们之前定义的损失函数，实际是单个样本的损失函数：

$$\text{LOSS} = (w(x_1 + x_2) + b - \text{MAX}(x_1, x_2))^2$$

当我们不断选择随机的样本，在幕后最小化的函数，是单个样本的损失函数在全体样本上的积分，并且对于这里的问题可计算出积分的结果：

$$\begin{aligned} & \int_0^1 \int_0^1 (w(x_1 + x_2) + b - \text{MAX}(x_1, x_2))^2 dx_1 dx_2 \\ &= \frac{1}{6}(3 - 8b + 6b^2 - 9w + 12bw + 7w^2) \end{aligned}$$

将它分别对 w 和 b 求导，并令导数都为 0，即可知在极值点处 $w = \frac{1}{2}$ ， $b = \frac{1}{6}$ 。

2.1.5 训练：Excel 的实现

我们可用 Excel 验证上述过程。首先输入表格，如表 2-2 所示。

表 2-2 要输入的表

行 / 列	A	B	C	D	E	F	G	H	I	J
1	t	x_1	x_2	目标	w_1	w_2	b	OUT	MSE	eta
2	0	=RAND()	=RAND()	=MAX(B2,C2)	0	0	0	=B2*E2+C2 *F2+G2	=(H2-D2) *(H2-D2)	=0.2*POWER (0.997, A2)
3	1									

其中 t 代表当前训练的样本编号。学习速率 eta 由 0.2 开始，并在每个样本后减少到原来的 0.997 倍，这是经实验效果较好的方案。

然后填入参数的训练公式：

□ 在 E3 单元格填入 $=E2 - J2 * 2 * (H2 - D2) * B2$ 。

□ 在 F3 单元格填入 $=F2 - J2 * 2 * (H2 - D2) * C2$ 。

□ 在 G3 单元格填入 $=G2 - J2 * 2 * (H2 - D2)$ 。

将表格下拉几百行完成填充，效果如图 2-8 所示。

	A	B	C	D	E	F	G	H	I	J
1	t	x1	x2	目标	w1	w2	b	OUT	MSE	eta
2	0	0.69	0.61	0.69	0.00	0.00	0.00	0.00	0.4770	0.200
3	1	0.12	0.84	0.84	0.19	0.17	0.28	0.44	0.1575	0.199
4	2	0.60	0.30	0.60	0.21	0.30	0.43	0.65	0.0025	0.199
5	3	0.84	0.28	0.84	0.20	0.29	0.41	0.66	0.0310	0.198
6	4	0.92	0.57	0.92	0.26	0.31	0.48	0.90	0.0004	0.198
7	5	0.73	0.79	0.79	0.26	0.32	0.49	0.94	0.0206	0.197
8	6	0.09	0.58	0.58	0.22	0.27	0.44	0.62	0.0012	0.196
9	7	0.61	0.34	0.61	0.22	0.27	0.42	0.65	0.0012	0.196
10	8	0.60	0.51	0.60	0.21	0.26	0.41	0.67	0.0053	0.195

图 2-8 Excel 中的训练过程

可在 Excel 中插入数据图，显示如图 2-9 所示的 w_1 、 w_2 、 b 、MSE 演变过程。

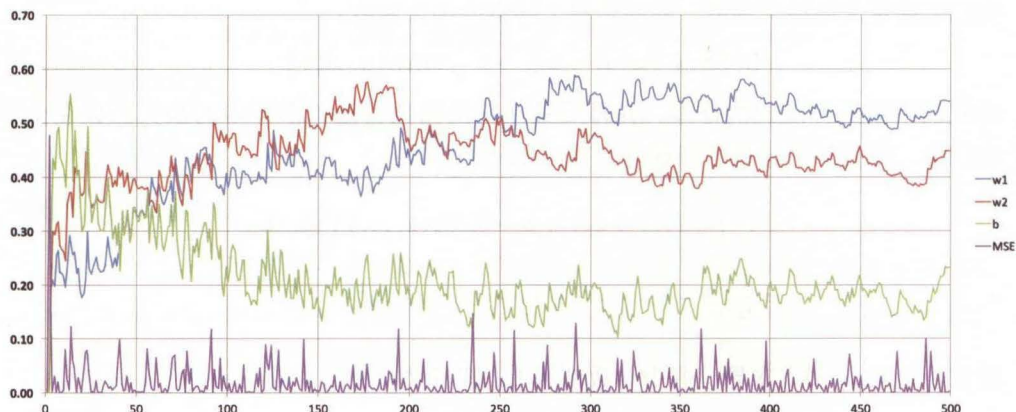


图 2-9 训练过程中参数的演变

2.1.6 重要知识：批大小、mini-batch、epoch

在之前的训练过程中，我们每次只选取 1 个样本，然后根据运行结果调整参数，这实际正是著名的随机梯度下降 (SGD)，而且可称为批大小 (batch size) 为 1 的 SGD。

批大小，就是每次调整参数前所选取的样本 (称为 mini-batch 或 batch) 数量：

- ❑ 如果批大小为 N ，每次会选取 N 个样本，分别代入网络，算出它们分别对应的参数调整值，然后将所有调整值取平均，作为最后的调整值，以此调整网络的参数。
- ❑ 如果批大小 N 很大，例如和全部样本的个数一样，那么可保证得到的调整值很稳定，是最能让全体样本受益的改变。
- ❑ 如果批大小 N 较小，例如为 1，那么得到的调整值有一定的随机性，因为对于某个样本最有效的调整，对于另一个样本不一定最有效 (就像对于识别某张黑猫图像最有效的调整，不一定对于识别另一张白猫图像最有效)。

那么批大小是否越大越好？绝非如此，很多时候恰好相反。合适的批大小对于网络的训练很重要，后文会介绍。

训练中的另一个重要概念是 epoch。每学一遍数据集，就称为 1 个 epoch。

举例，若数据集中有 1000 个样本，批大小为 10，那么将全部样本训练 1 遍后，网络会被调整 $1000/10=100$ 次。但这并不意味着网络已达到最优，我们可重复这个过程，让网络再学 1 遍、2 遍、3 遍数据集。

注意每一个 epoch 都需打乱数据的顺序，以使网络受到的调整更具有多样性。同时，我们会不断监督网络的训练效果。通常情况下，网络的性能提高速度会越来越慢，在几十到几百个 epoch 后网络的性能会趋于稳定，即性能基本不再提高。

2.2 深度学习框架 MXNet：安装和使用

深度学习编程往往会通过深度学习框架 (framework)，深度学习框架类似于函数库。如何选择深度学习框架？对于个人爱好者而言，训练网络往往是个漫长的过程，因为我们的训练硬件资源有限。因此，选择一个训练速度快的框架很重要。此外，框架本身的流行程度和易用程度也很重要。

常用深度学习框架在 GitHub 上的热度，以及在它们之上运行 2 种典型网络 (AlexNet 和 ResNet-50) 的训练耗时^①，如表 2-3 所示。

表中 PyTorch、Chainer、Keras 的运行速度暂未在论文中测试。据经验，其中 Keras 和 Chainer 较慢，PyTorch 的速度较好。由表可见：

- ❑ Google 的 TensorFlow 是目前最热门的框架，适合大规模集群部署。缺点是训练速度较慢，使用和调试都有些繁琐。

① 根据 <https://arxiv.org/pdf/1608.07249.pdf> 的测试结果。

表 2-3 训练耗时表

名称	GitHub 主页	GitHub Star 数	GitHub 贡献者数	AlexNet 耗时	ResNet-50 耗时
TensorFlow	https://github.com/tensorflow/tensorflow	67 396	997	0.042	0.227
MXNet	https://github.com/apache/incubator-mxnet	10 800	407	0.014	0.136
CNTK	https://github.com/Microsoft/CNTK	12 121	145	0.032	0.168
Caffe	https://github.com/BVLC/caffe	19 720	248	0.021	0.254
Torch	https://github.com/torch/torch7	7192	132	0.023	0.144
PyTorch	https://github.com/pytorch/pytorch	6800	280		
Chainer	https://github.com/chainer/chainer	2830	132		
Keras	https://github.com/fchollet/keras	18 795	509		

❑ MXNet 的训练速度最快，代码结构清晰，同时在 GitHub 上的贡献者也较多，版本更新快。值得一提的是，MXNet 的开发者大多为中国人，同时技术更胜于许多国外框架。其实，在目前许多 AI 学术会议中，中国人的面孔也已越来越多。

❑ Microsoft 的 CNTK 有一些特别的功能，如支持 C#、支持 1-bit 网络。

❑ Caffe 和 Torch 是经典的框架，都来自于 Facebook，目前用户有往其他框架流失的趋势。此外还有 Theano，它是最早的框架，目前用户也在减少之中。

❑ Facebook 的 PyTorch 诞生于 2017 年，采用动态图计算（后文会解释），适合学术研究，有不少开发者选择。

○ MXNet 在最新版本也推出了动态图支持（称为 Gluon），且速度更快。感兴趣的读者可参阅 <http://gluon.mxnet.io/>。

○ TensorFlow 在最近也推出了动态图支持（称为 Eager 模式）。

❑ Chainer 来自日本，是动态图计算的先行者，有不少二次元（动漫）的例子，感兴趣的读者可查询。

❑ Keras 并非独立的深度学习框架，而是现有框架之上的封装，希望让现有的框架更易使用，同时更易在不同框架之间转换。缺点是运行速度更慢，且不够灵活。它可采用 TensorFlow 或 CNTK 等作为后端运行引擎，目前 MXNet 也对其有一定的支持。

建议读者安装多个深度学习框架，因为在 GitHub 可搜索到许多深度网络模型代码，但使用的框架各异。如果安装了多个深度学习框架，就可以直接运行这些制作好的模型。

如果读者有能力，甚至可自己写一个深度学习框架，以实现更快的速度、更小的资源占用、更灵活的运作。对此感兴趣的读者可阅读 4.5 节。

2.2.1 计算图：动态与静态

前面涉及动态图和静态图这两个概念，在此解释。由于训练神经网络会涉及求导数，因此深度学习框架都会构造“计算图”（computation graph）。

例如，若 $e=(a+b) \times (b+1)$ ，则可构造出如图 2-10 所示的计算图，其实就是把所有中间步骤都写出来。

在 TensorFlow、Caffe、MXNet 等静态图框架中，计算图的构造方式是：先构造，再运行，运行的时候不能再修改。换言之，先定义网络模型，然后送入执行器执行。这类似于编程语言的“先编译，再运行”，优势是理论运行速度快，可以做更多优化；缺点是运行时难以修改计算图的构造，因此有时不够灵活。不过，通常的神经网络模型并不需要在运行时修改，除非是做研究，需要实验特别的架构。

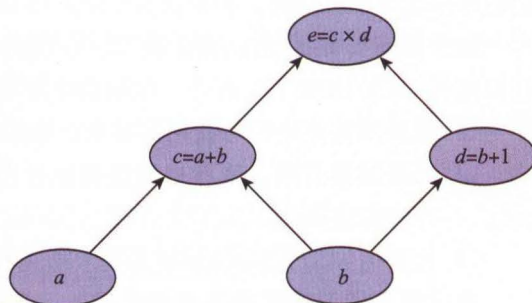


图 2-10 计算图

而在 PyTorch、Chainer、MXNet Gluon 等动态图框架中，计算图是动态构造，根据程序语句动态生成，类似于编程语言的解释执行，程序写起来更灵活方便，缺点是理论运行速度会相对较慢。MXNet Gluon 提供了类似于编译的功能，可解决这个问题。

静态图框架，也可称为符号式 (symbolic) 框架。动态图框架，也可称为解释式 (imperative) 框架。常见框架的情况如图 2-11 所示 (其中红色 K 是 Keras)。

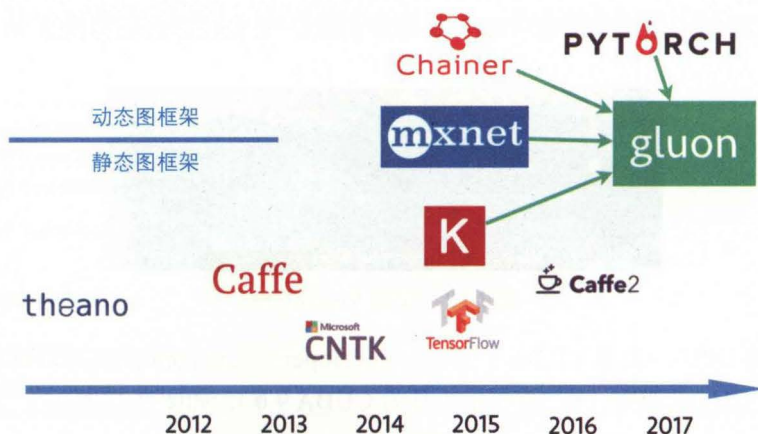


图 2-11 动态图框架与静态图框架

2.2.2 安装 MXNet : 准备工作

下面我们安装 MXNet 框架。由于 MXNet 的版本会不断更新，如果读者在安装时遇到问题，可访问 MXNet 主页 (<http://mxnet.io/>) 查看最新安装指南。

如果读者有 nVidia 的显卡，安装时可选择 GPU 版本框架。这需要预先安装 2 个 nVidia 公司的加速库——CUDA 和 cuDNN，它们可提供在 nVidia GPU 上高速运行和训练深度神经网络所需的函数支持。

最近 AMD 也提供了与之相对应的加速库，称为 ROCm 和 MIOpen，不过截至 2017 年，它们仍在测试之中。因此，如果读者希望采用 GPU 加速深度学习，目前推荐使用 nVidia 显卡。

如果读者计划购置 nVidia 显卡，在现阶段（2017 年），推荐至少选择 GTX 1070 起步，建议选择 GTX 1080 Ti。此外，电脑的电源功率要配足，电脑内存也推荐配到 16G 或以上。

- ❑ 如果预算充足，可使用多显卡，请注意选择 PCI-E 通道数足够多的 CPU。
- ❑ 如果预算有限，推荐尽量选择显存在 8G 以上的 nVidia 显卡，因为深度学习对于显存的需求很大。
- ❑ 可采用 MSI Afterburner 为显卡超频和降压。

决定深度学习速度的是单精度浮点运算速度（GFLOPs）。由表 2-4 所示可见 GPU 远远快于 CPU，而 Google 的 TPU v2 更快。

表 2-4 运算速度

设备	Core i7-7700K	GTX 1080 Ti	Google TPU v2
GFLOPs	538	11 340	180 000

2.2.3 在 Windows 下安装 MXNet

推荐使用 Windows 10，训练速度更快。安装时注意在管理员权限下执行各个程序。

1) 虽并非必须，但推荐安装 Visual Studio 2015，安装时语言记得选上 Visual C++，如图 2-12 所示。

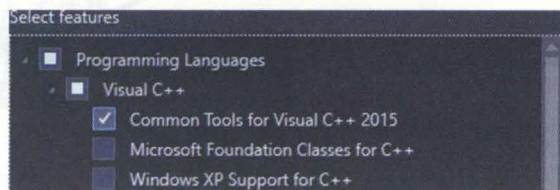


图 2-12 安装 Visual C++

2) 如果有 GPU，安装 CUDA：<https://developer.nvidia.com/cuda-downloads>。请选择 MXNet 支持的 CUDA 版本（在 2018 年 3 月为 CUDA 9.0），如图 2-13 所示。

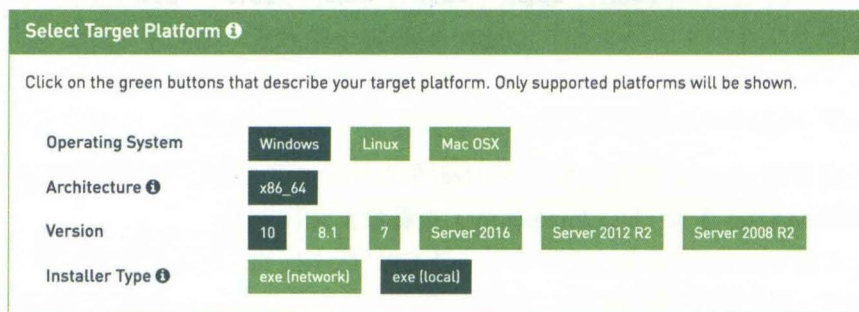


图 2-13 安装 CUDA

注意选本地安装版，即 exe[local]。如需下载旧版本的 CUDA，可选择此页面下方的 Legacy Releases。

之后可直接用 `pip install mxnet-cu90` 安装 GPU 版本 MXNet，或用 `pip install mxnet` 安装 CPU 版本。如需手动安装，可见下文的 3) 到 7)。

3) 如果有 GPU，下载 cuDNN：<https://developer.nvidia.com/cudnn>。下载前需注册 nVidia 账号，如图 2-14 所示。

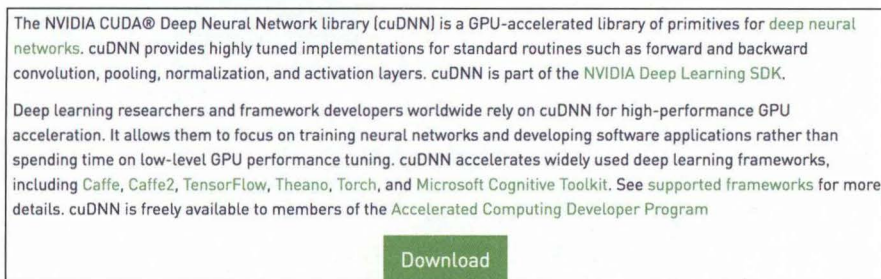


图 2-14 安装 cuDNN

下载后解压，将解压后的文件移动到之前安装的 CUDA 目录（会看到有一致的目录结构）。

4) 下载 MXNet 的预编译包：<https://github.com/yajiedesign/mxnet/releases>，如图 2-15 所示。



图 2-15 下载 MXNet

先下载基础包（vc14 base package），然后下载更新包，如 2017****_mxnet_x64_vc14_gpu.7z（如果有 GPU，选 GPU 版，会同时支持 CPU 和 GPU）。将基础包和更新包解压到同一个目录。

5) 如果有 GPU，把第 3 步解压的 cuDNN 复制一份到第 4 步解压的目录下的 3rdparty 目录下的 cudnn 目录。

6) 安装 Python，推荐 Anaconda：<https://www.continuum.io/downloads>。注意目前的 MXNet 需选择 Python 2.7 版本，如图 2-16 所示。

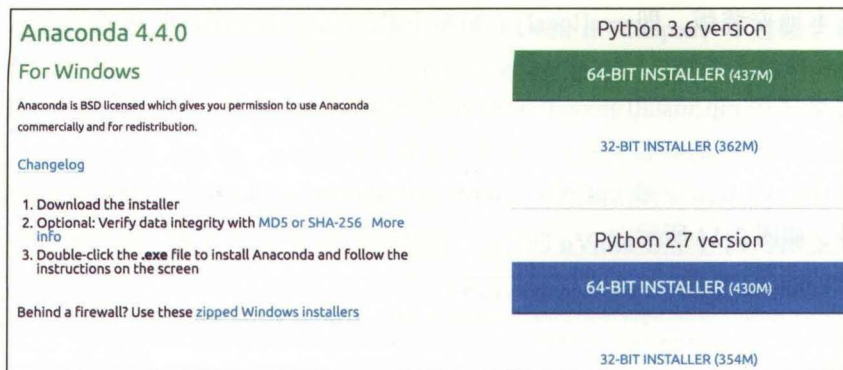


图 2-16 安装 Python

安装后请在 Windows 命令行检验 Python 命令是否可用。

7) 进入之前解压的 MXNet 目录，在命令行执行：

```
setupenv.cmd
```

完成设置环境。然后进入 python 子目录，在命令行执行：

```
python setup.py install
```

如提示需安装 Microsoft Visual C++ Compiler for Python 2.7，可在如下地址下载并安装：<https://www.microsoft.com/en-us/download/details.aspx?id=44266>。

如提示缺头文件，将它们从 VC2015 的 include 目录复制到 Microsoft Visual C++ Compiler for Python 2.7 的 include 目录。可能需根据提示复制多个头文件，然后即可成功编译。

8) 运行 Python，然后在 Python 中执行：

```
import mxnet
(mxnet.nd.ones((2,2), mxnet.cpu())*100).asnumpy() # 测试 CPU 运算
(mxnet.nd.ones((2,2), mxnet.gpu())*100).asnumpy() # 测试 GPU 运算
```

如果此前全部安装成功，会看到这样的输出：

```
array([[ 100.,  100.],
       [ 100.,  100.]], dtype=float32)
```

9) 测试训练神经网络。下载 MXNet 的完整代码：<https://github.com/dmlc/mxnet>。解压后，进入 example 目录的 image-classification 目录，在命令行执行：

```
python train_mnist.py --network lenet
```

这是用 LeNet-5 网络训练 MNIST 数据集，这是经典的神经网络例子。它会先下载 MNIST 数据，由于数据在国外，下载速度较慢，可耐心等待。

如果运行成功，可看到准确率 (Train-accuracy) 随着训练不断提高，如图 2-17 所示。

10) 安装 GraphViz：<http://graphviz.org>。将安装目录下的 bin 目录加入 PATH 环境变量，以保证可在命令行直接运行 dot 命令。


```

管理工具: C:\Windows\system32\cmd.exe - python train_mnist.py --network lenet
DEBUG:requests.packages.urllib3.connectionpool:"GET /exdb/mnist/train-images-idx3-ubyte.gz HTTP/1.1" 200 9912422
INFO:requests.packages.urllib3.connectionpool:Starting new HTTP connection (1): yann.lecun.c
on
DEBUG:requests.packages.urllib3.connectionpool:"GET /exdb/mnist/t10k-labels-idx1-ubyte.gz HTTP/1.1" 200 4542
INFO:requests.packages.urllib3.connectionpool:Starting new HTTP connection (1): yann.lecun.c
on
DEBUG:requests.packages.urllib3.connectionpool:"GET /exdb/mnist/t10k-images-idx3-ubyte.gz HTTP/1.1" 200 1648877
INFO:root:Epoch[0] Batch [100] Speed: 242.02 samples/sec Train-accuracy=0.854889
INFO:root:Epoch[0] Batch [200] Speed: 237.83 samples/sec Train-accuracy=0.948281
INFO:root:Epoch[0] Batch [300] Speed: 232.03 samples/sec Train-accuracy=0.960313
INFO:root:Epoch[0] Batch [400] Speed: 226.79 samples/sec Train-accuracy=0.967031
INFO:root:Epoch[0] Batch [500] Speed: 225.84 samples/sec Train-accuracy=0.970469
INFO:root:Epoch[0] Batch [600] Speed: 231.52 samples/sec Train-accuracy=0.976719
INFO:root:Epoch[0] Batch [700] Speed: 229.80 samples/sec Train-accuracy=0.972344
INFO:root:Epoch[0] Batch [800] Speed: 239.31 samples/sec Train-accuracy=0.973906
INFO:root:Epoch[0] Batch [900] Speed: 237.71 samples/sec Train-accuracy=0.979062
INFO:root:Epoch[0] Train-accuracy=0.979307
INFO:root:Epoch[0] Time cost=258.056
INFO:root:Epoch[0] Validation-accuracy=0.977408
INFO:root:Epoch[1] Batch [100] Speed: 234.00 samples/sec Train-accuracy=0.980198
INFO:root:Epoch[1] Batch [200] Speed: 204.21 samples/sec Train-accuracy=0.981094
INFO:root:Epoch[1] Batch [300] Speed: 190.04 samples/sec Train-accuracy=0.984219
INFO:root:Epoch[1] Batch [400] Speed: 222.15 samples/sec Train-accuracy=0.982344
INFO:root:Epoch[1] Batch [500] Speed: 215.42 samples/sec Train-accuracy=0.986563
INFO:root:Epoch[1] Batch [600] Speed: 233.75 samples/sec Train-accuracy=0.985938
INFO:root:Epoch[1] Batch [700] Speed: 236.94 samples/sec Train-accuracy=0.985000

```

图 2-17 MNIST 的训练过程

2.2.4 在 macOS 下安装 MXNet : CPU 版

下面我们看 macOS 下 MXNet 的安装。CPU 版的安装较为简单，因为已有编译好的包。

1) 安装 Homebrew。可在百度搜索方法，或执行：

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

2) 安装 Python、Graphviz 及 MXNet（注意 pip 指令前可能需加上 sudo -H）：

```

brew install python
brew install graphviz
pip install graphviz
brew install mxnet
pip install mxnet

```

由于需要下载的许多文件在国外，下载速度可能较慢。可改成使用阿里的 pip 源：

```

pip install 要安装的库 --upgrade -i http://mirrors.aliyun.com/pypi/simple
--trusted-host mirrors.aliyun.com

```

并可在下载前设置使用 Homebrew 的国内源：

```

cd "$(brew --repo)"
git remote set-url origin https://mirrors.ustc.edu.cn/brew.git
cd "$(brew --repo)/Library/Taps/homebrew/homebrew-core"
git remote set-url origin https://mirrors.ustc.edu.cn/homebrew-core.git
cd "$(brew --repo)/Library/Taps/caskroom/homebrew-cask"
git remote set-url origin https://mirrors.ustc.edu.cn/homebrew-cask.git
echo 'export HOMEBREW_BOTTLE_DOMAIN=https://mirrors.ustc.edu.cn/homebrew-bottles'

```

```
>> ~/.bash_profile
source ~/.bash_profile
```

3) 进入 Python, 与前文类似, 检查是否安装成功。

2.2.5 在 macOS 下安装 MXNet : GPU 版

如果读者的 Mac 有 nVidia 显卡 (注意 MacBook 可通过外接雷电显卡盒使用 nVidia 显卡), 或希望自行编译 MXNet, 在此我们也给出步骤。这个过程中如遇到问题, 可在 Google 搜索, 或在 MXNet 的 GitHub 页面的 Issues 栏目发帖询问。

安装时可能会遇到权限问题, 可通过 chown 解决。有时也可能需 sudo。由于需下载的许多文件在国外, 下载速度可能较慢。可在命令行设置代理服务器:

```
export HTTP_PROXY="代理服务器地址"
export HTTPS_PROXY="代理服务器地址"
export ALL_PROXY="代理服务器地址"
```

此外 git 也可加上代理服务器:

```
git config --global http.proxy '代理服务器地址'
git config --global https.proxy '代理服务器地址'
```

1) 安装 XCode。然后按前文方法, 下载和安装 CUDA 和 cuDNN。再进入系统设置面板, 选择 CUDA, 更新驱动。另外做一个符号链接:

```
ln -s /usr/local/cuda/lib /usr/local/cuda/lib64
```

2) 如上节所述, 安装 Homebrew。然后安装 Python, Graphviz 以及 OpenBLAS:

```
brew install python
brew install graphviz
pip install graphviz
brew install openblas
```

3) 将 MXNet 下载到 ~/mxnet 目录:

```
git clone https://github.com/dmlc/mxnet.git ~/mxnet --recursive
```

4) 进入 ~/mxnet 目录, 执行编译指令 (这里为方便阅读做了分行, 实际应全部写在同一行):

```
make -j 4 USE_OPENCV=0 USE_BLAS=openblas USE_CUDA=1 USE_CUDNN=1
USE_NVRTC=1 USE_PROFILER=1 USE_OPENMP=0
USE_CUDA_PATH=/usr/local/cuda
ADD_CFLAGS+="-I/usr/local/opt/openblas/include
ADD_LDFLAGS+="-L/usr/local/opt/openblas/lib
ADD_LDFLAGS+="-L/usr/local/lib/graphviz/
```

其中 -j 4 代表用 4 核编译。

5) 如出现如下错误:


```
nvcc fatal : The version xxx of the host compiler ('Apple clang') is not supported
```

说明 XCode 的版本太高或 CUDA 的版本不够，互相还没有兼容，可在网上搜索 XCode Command Line Tools 的降级方法。例如，可在 <https://developer.apple.com/download/more/> 处下载旧版本的 CLT，然后执行 `sudo xcode-select --switch /Applications/Xcode.app/Contents/Developer`，然后执行 `clang --version` 看版本号是否降低。

6) 编译完成后，进入 `python` 子目录，完成安装：

```
sudo python setup.py install
```

如果出现错误，很可能是因为之前没有编译成功，通常是 CUDA 的编译问题。可能需 `make clean_all` 清理后再试试。

2.2.6 在 Linux 下安装 MXNet

我们可通过与 macOS 类似的方法编译 MXNet，也可直接用 `pip` 安装。

安装 CPU 最新版方法：

```
pip install mxnet --pre --user
```

安装 GPU 最新版（并使用 CUDA 8.0）方法：

```
pip install mxnet-cu80 --pre --user
```

关于 Linux 下安装 CUDA 的方法，可参照 <http://docs.nvidia.com/cuda/cuda-installation-guide-linux/>。

2.2.7 安装 Jupyter 演算本

Jupyter 是一个演算本环境，可在其中运行 Python 代码，并直观地显示 Python 程序的输出，类似于 Mathematica 的效果。

可通过 `pip` 安装 Jupyter，方法是在安装 Python 后，执行：

```
pip install jupyter
```

然后，在命令行运行：

```
jupyter notebook
```

会打开一个浏览器窗口，显示当前目录中的文件，如图 2-18 所示。

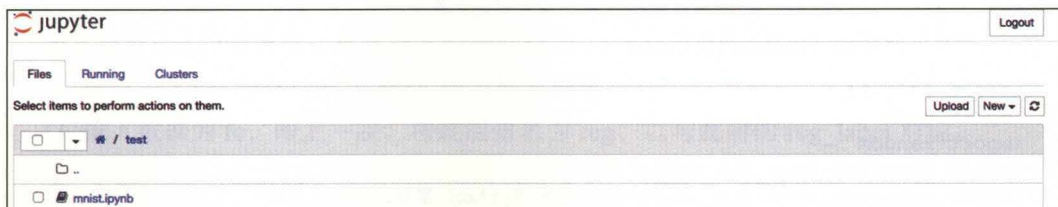


图 2-18 Jupyter 的目录界面

建议建立一个专用于存放 Jupyter 演算本的目录，然后每次先在命令行进入这个目录，再执行 jupyter notebook 指令，这就可选择目录中的演算本文件。

Jupyter 演算本的后缀名是 ipynb。在 Jupyter 的浏览器界面中可新建 ipynb，也可点击现有的 ipynb 进入演算本的界面，如图 2-19 所示。

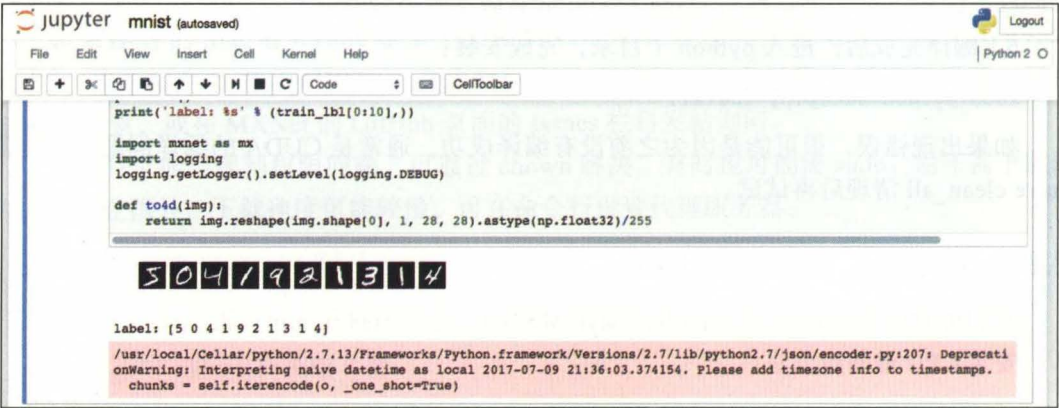


图 2-19 Jupyter 的演算本界面

在演算本中，可直观地运行程序的每个片段，看到程序的每个片段的输出。

2.2.8 实例：在 MXNet 训练神经元并体验调参

在此我们使用 MXNet 训练之前的神经元。程序的架构如图 2-20 所示。

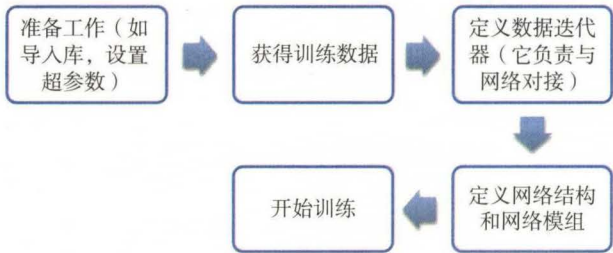


图 2-20 训练程序的架构

下面看代码。首先，导入需要的库：

```

#-*-coding:utf-8-*- # 如果使用py文件，需声明此文件为 UTF-8 格式，这样可以使用中文注释。如
    果使用ipynb文件，则无须此行

import logging
import math
import random
import mxnet as mx # 导入 MXNet 库
import numpy as np # 导入 NumPy 库，这是 Python 常用的科学计算库

logging.getLogger().setLevel(logging.DEBUG) # 打开调试信息的显示

```


设置超参数：

```
n_sample = 1000 # 训练用的数据点个数
batch_size = 1 # 批大小
learning_rate = 0.1 # 学习速率
n_epoch = 1 # 训练 epoch 数
```

生成训练数据：

```
# 每组数据是在 (0,1) 之间的 2 个随机数，共 n_sample 组数据
train_in = [[ random.uniform(0, 1) for c in range(2)] for n in range(n_sample)]

train_out = [0 for n in range(n_sample)] # 期望输出，先初始化为 0

for i in range(n_sample):
    # 每组数据的期望输出，是 2 个输入数中的大者
    train_out[i] = max(train_in[i][0], train_in[i][1])
```

定义变量 `train_iter` 为训练数据的迭代器 (Iter)，它将负责提供数据给训练过程。在此我们使用 MXNet 内置的数组迭代器 (NDArrayIter)。

- ❑ 将输入数据 (data) 设置为转换为 Numpy 数组的 `train_in`。
- ❑ 这里的 `np.array` 函数可将 Python 数组转为 NumPy 数组。MXNet 经常会使用 NumPy 数组。
- ❑ 将期望输出 (称为 label，即标签) 命名为 `reg_label`，并与 `train_out` 对接。
- ❑ 将 `shuffle` 设置为真 (True)，代表每个 epoch 会随机打乱数据，这对于提高训练效果很重要。

```
train_iter = mx.io.NDArrayIter(data = np.array(train_in), label = {'reg_label': np.array(train_out)}, batch_size = batch_size, shuffle = True)
```

接着定义网络结构。定义变量 `src` 为输入层，具体内容是 MXNet 的内部变量 'data'，代表训练数据。

```
src = mx.sym.Variable('data')
```

定义变量 `fc` 为全连接层 (FullyConnected)。

- ❑ 输入数据 (data) 为变量 `src`，即输入层，即训练数据。
- ❑ 将 `num_hidden` 设置为 1，代表此层有 1 个神经元。
- ❑ 命名 (name) 为 `fc`，以方便后续 MXNet 辨认。建议为每一层设置唯一的名字。

```
fc = mx.sym.FullyConnected(data = src, num_hidden = 1, name = 'fc')
```

定义变量 `net` 为输出层，具体是线性回归输出 (LinearRegressionOutput)。

- ❑ 此时 MXNet 会自动使用 MSE 作为损失函数。
- ❑ 输入数据为 `fc`，即上一层。将此层命名为 `reg`，与此前的 `reg_label` 对应。

```
net = mx.sym.LinearRegressionOutput(data = fc, name = 'reg')
```

定义变量 `module` 为需训练的网络模组。

- 网络的输出 (symbol) 为刚才定义的 net 变量。
- 期望标签名 (label_names) 为 reg_label, 与此前对应。

```
module = mx.mod.Module(symbol = net, label_names = (['reg_label']))
```

定义 epoch_callback 函数, 在此它负责输出训练出的网络的参数值。

```
def epoch_callback(epoch, symbol, arg_params, aux_params):
    for k in arg_params: # 对于所有网络的参数...
        print(k) # 输出参数名
        print(arg_params[k].asnumpy()) # 输出参数值, 这里转为 NumPy 数组, 输出更美观
```

最后, 调用 module.fit, 开始训练:

```
module.fit(
    train_iter, # 训练数据的迭代器
    eval_data = None, # 在此只训练, 不使用测试数据
    eval_metric = mx.metric.create('mse'), # 输出 MSE 损失信息
    optimizer = 'sgd', # 梯度下降算法为 SGD
    # 设置学习速率
    optimizer_params = {'learning_rate': learning_rate},
    num_epoch = n_epoch, # 训练 epoch 数
    # 每经过 100 个 batch 输出训练速度
    batch_end_callback = mx.callback.Speedometer(batch_size, 100),
    epoch_end_callback = epoch_callback, # 完成每个 epoch 后调用 epoch_callback
)
```

这里没有指定神经网络参数的初始化方法, MXNet 默认会用较小的随机数初始化网络。

程序的输出样例如下。其中 Speed 代表训练速度。默认情况下 MXNet 会使用 CPU 训练, 所以训练速度并不是很快:

```
INFO:root:Epoch[0] Batch [100] Speed: 3136.96 samples/sec mse=0.039371
INFO:root:Epoch[0] Batch [200] Speed: 3100.87 samples/sec mse=0.017533
INFO:root:Epoch[0] Batch [300] Speed: 3048.87 samples/sec mse=0.015845
INFO:root:Epoch[0] Batch [400] Speed: 3235.72 samples/sec mse=0.014175
INFO:root:Epoch[0] Batch [500] Speed: 3052.69 samples/sec mse=0.018520
INFO:root:Epoch[0] Batch [600] Speed: 3185.42 samples/sec mse=0.016790
INFO:root:Epoch[0] Batch [700] Speed: 3050.91 samples/sec mse=0.016181
INFO:root:Epoch[0] Batch [800] Speed: 3200.00 samples/sec mse=0.012819
INFO:root:Epoch[0] Batch [900] Speed: 2981.53 samples/sec mse=0.016120
INFO:root:Epoch[0] Train-mse=0.012219
INFO:root:Epoch[0] Time cost=0.323
fc_bias
[ 0.1689797]
fc_weight
[[ 0.46135306 0.50179851]]
```

可见训练出的偏置在 $\frac{1}{6}$ 左右, 权重在 $\frac{1}{2}$ 左右, 与前文一致。同时, MSE 在下降到 $\frac{1}{72}$ 左右后就不会继续下降, 说明单个神经元无法完美地完成这个任务。

由于神经网络的初始化是随机的, 训练数据也是随机的, 每次运行将得到不同的结果。有时 MXNet 在 Windows 下会出现程序运行完毕无法退出的情况, 如果读者遇到这种

情况，可手动在任务管理器结束 Python 进程即可。

下面我们试试调整超参数 (hyper-parameters)，即著名的“调参”。

- 网络的参数，是指网络的权重和偏置。
- 网络的超参数，包括学习速率、批大小、网络架构（如每层的神经元数目）等。
- 设置合理的超参数，称为“调参”，可让网络达到更好的训练效果。

在此读者可实验如下的调参方法：

- 将训练样本数增大（例如设置为 10000），同时将学习速率降低（例如设置为 0.01）。会发现每个 epoch 后可得到更准确的值，但训练过程会变慢。
- 于是可再将批大小加大（例如设置为 10），然后把学习速率提高（例如提高回 0.1）。会发现训练速度明显更快，同时也很准确。

2.3 神经网络：运作和训练

在前文我们看到，单个神经元的能力有限，甚至无法保证找到两个数中的更大者。但通过多个神经元，就可完成这个目标。神经元可相互将输出和输入连接，构成神经网络，如图 2-21 所示。

这个网络的情况如下：

- 有 2 个输入，即图中的 x_1 、 x_2 。
- 有 3 个神经元，即图中的 b_1 、 b_2 、 b_3 。
 - 为方便起见，直接标记它们的偏置在圆圈中，为 b_1 、 b_2 、 b_3 。
 - 为方便起见，直接将这 3 个神经元命名为 b_1 、 b_2 、 b_3 。

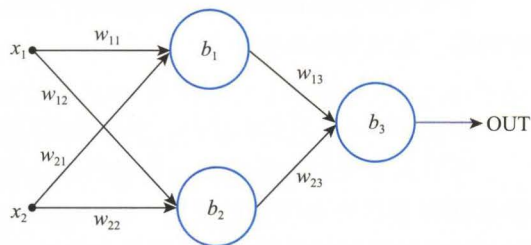


图 2-21 神经网络实例

- 共有 2 层网络。第 1 层的神经元是 b_1 、 b_2 ，第 2 层的神经元是 b_3 。可将中间层称为隐层 (hidden layer)。例如图中的 b_1 、 b_2 就属于隐层。
- 有 6 个权重 (w_{11} 到 w_{23})。最终输出为 OUT。读者可能会担心这里将从 x_i 到 b_j 的权重，与从 b_i 到 b_j 的权重都标为 w_{ij} ，是否会容易混淆？不过如果仔细想想会发现并不会出现冲突的情况。

值得特别注意的是，在实际的神经网络中，神经元之间并不是直接连接，需要插入一个构造：非线性激活 (non-linear activation)，这在下文介绍。

2.3.1 运作：前向传播，与非线性激活的必要性

神经网络的运作过程可称为“前向传播”(forward-propagation)。

为说明非线性激活的必要性，不妨先看如果没有非线性激活会是怎样。以刚才的网络为例：

- 神经元 b_1 的输出是 $x_1 w_{11} + x_2 w_{21} + b_1$ 。
- 神经元 b_2 的输出是 $x_1 w_{12} + x_2 w_{22} + b_2$ 。
- 如果没有非线性激活，那么最终输出 OUT 的计算公式是：

$$\begin{aligned}\text{OUT} &= (x_1 w_{11} + x_2 w_{21} + b_1) \cdot w_{13} + (x_1 w_{12} + x_2 w_{22} + b_2) \cdot w_{23} + b_3 \\ &= x_1 \cdot (w_{11} w_{13} + w_{12} w_{23}) + x_2 \cdot (w_{21} w_{13} + w_{22} w_{23}) + (b_1 w_{13} + b_2 w_{23} + b_3)\end{aligned}$$

换言之，虽然使用了 3 个神经元，但这个网络对于 x_1 和 x_2 仍然是线性的，完全等价于 1 个神经元的效果，如图 2-22 所示。

于是这就失去了使用多个神经元的意义。这个现象可用数学解释：神经元的输出是输入的线性函数，而线性函数之间的嵌套仍然会得到线性函数，就像矩阵相乘仍然会得到矩阵。因此，如果只是把神经元简单连接在一起，不加入非线性处理，那么最终得到的仍然是线性函数，这就无法描述各种复杂的现象。

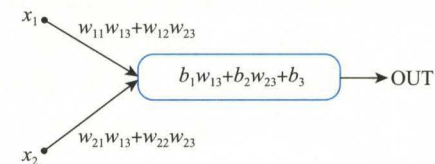


图 2-22 线性神经网络等价于线性神经元

为此，我们会在神经元的输出后加入非线性激活，即，将神经元的输出由 t 变为 $f(t)$ ，其中 f 是一个非线性函数，不同的神经元可使用不同的 f 。

若非线性激活函数对于中间层神经元是 f ，对于最终输出神经元是 g ，那么，对于此前的网络，最终的输出会变为：

$$\text{OUT} = g(f(x_1 w_{11} + x_2 w_{21} + b_1) \cdot w_{13} + f(x_1 w_{12} + x_2 w_{22} + b_2) \cdot w_{23} + b_3)$$

由于 f 和 g 都是非线性的，于是最终可得到非线性的网络输出。

在引入非线性激活之后，神经网络具有很强的表达能力，可拟合复杂的数据。通过数学可证明著名的通用逼近定理 (universal approximation theorem)，即此时只要有 1 个隐层和足够多的隐层神经元，就足以拟合任何函数。

但在实际神经网络中，往往会使用多个隐层，这就是深度神经网络，它不但可得到更好的训练效果，而且在合理设计后也会得到更好的实际测试效果。用术语说，就是更能克服欠拟合和过拟合。

最后，当批大小大于 1 时，我们可合并多个输入的前向传播过程，将前向传播过程写成矩阵的形式，这可实现更快的运行速度。在一定程度上，批大小越大，运行速度越快。

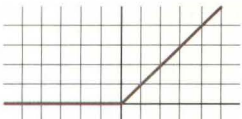
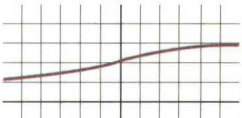
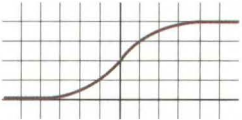
2.3.2 运作：非线性激活

读者可在 Wikipedia 上看到多种非线性激活函数的例子 (https://en.wikipedia.org/wiki/Activation_function)。在现代神经网络架构中，最常用的非线性激活函数是 ReLU、sigmoid (又称 logistic) 和 tanh，如表 2-5 所示。

建议有能力的读者验证表中的导数。其中 sigmoid 函数可写作 $\sigma(x)$ ，它的导数计算过程如下：

$$\sigma'(x) = \left(\frac{1}{1 + e^{-x}} \right)' = \frac{e^{-x}}{(1 + e^{-x})^2} = \frac{1}{1 + e^{-x}} \cdot \frac{e^{-x}}{1 + e^{-x}} = \sigma(x)(1 - \sigma(x))$$

表 2-5 常用非线性激活函数

名 称	图 像	定 义	导 数
ReLU		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Logistic(sigmoid)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$

这个计算过程很重要，值得掌握。

而这三者中最常用的非线性激活函数是 ReLU 函数，它有几个优点：

- ReLU 的运算非常简单、快速。
- ReLU 可避免“梯度消失”，稍后会介绍。
- 注意 ReLU 在 $x > 0$ 时才有非零输出，而在 $x \leq 0$ 时输出 0，所以 ReLU 更类似于生物神经元的工作原理，因为生物神经元需要一定的刺激才会被激活。
- 使用 ReLU 往往能带来比使用 sigmoid 和 tanh 时更佳的网络性能。

在引入 ReLU 后，此前的“输出两个 0 到 1 之间的数中的较大者”的问题，可通过简单的网络解决，如图 2-23 所示。

这是因为，如果 $0 < x < 1$ ， $0 < y < 1$ ，则

$$\text{MAX}(x, y) = \text{MAX}(0, x) + \text{MAX}(0, y - x)$$

读者可选几个数，并带入网络证实这一点。

在 ReLU 基础上，研究人员还提出了 Leaky ReLU、PReLU、RReLU、ELU、SELU 等 ReLU 的变种。它们会让 ReLU 在 $x < 0$ 时也拥有少量输出，在某些情况下可略微改进 ReLU 的性能，代价是计算

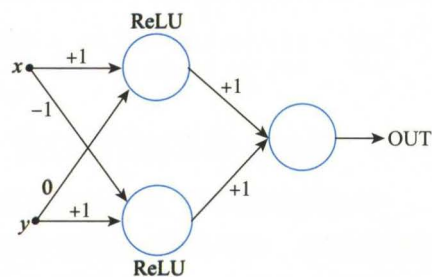


图 2-23 能输出两个 0 到 1 之间的数中的较大者的网络

速度会慢一些。这方面可参见 <https://arxiv.org/abs/1502.01852> (PReLU)、<https://arxiv.org/abs/1511.07289> (ELU) 和 <https://arxiv.org/abs/1706.02515> (SELU) 处的论文。

在 MXNet 中可直接调用各种非线性激活函数，而且 MXNet 会自动计算它们的导数，因此使用很简单，只需直接修改代码即可。

最后，在 2017 年 10 月，Google Brain 的研究人员经过大规模搜索 (<https://arxiv.org/>

abs/1710.05941), 发现了一种在许多任务上比 ReLU 性能更好的非线性激活函数, 称为 Swish: $f(x)=x \cdot \sigma(\beta x)$ 。其中 β 可固定为 1, 也可设置为是可训练的参数。

在 β 为 1 时, Swish 与 ReLU 的对比如图 2-24 所示。

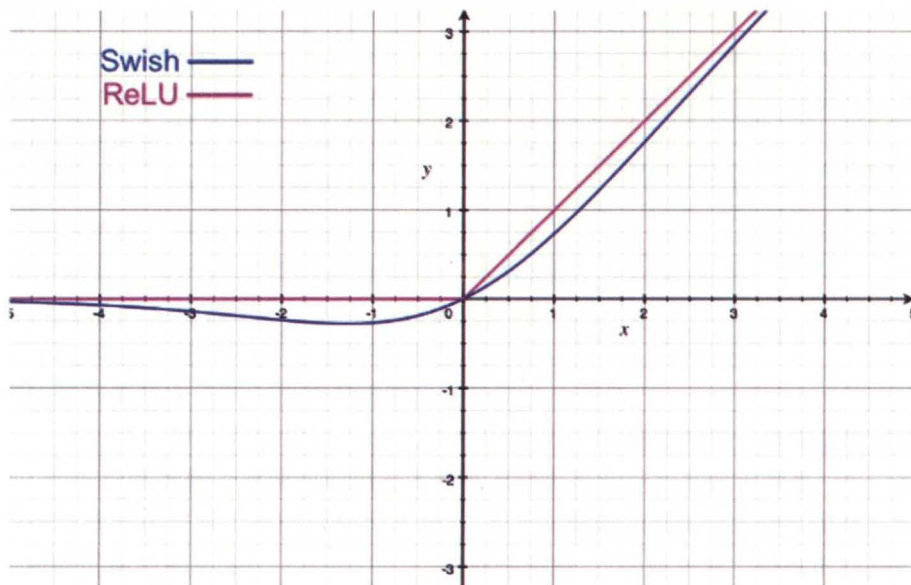


图 2-24 Swish 与 ReLU 的对比

由图可见, Swish 在 $x < 0$ 时是非单调的, 这有助于提高网络的非线性, 学会更复杂的数据。在 MXNet 使用 $\beta=1$ 的 Swish 的方法如下:

```
tmp = mx.sym.sigmoid(in) # 将输入做sigmoid处理
out = mx.sym.broadcast_mul(in, tmp) # 将它点对点乘以输入, 得到输出
```

2.3.3 训练：梯度的计算公式

神经网络的训练, 需要运用著名的“反向传播”(Back-Propagation, BP)。由于这个过程略为烦琐, 许多书籍在此的推导都较难理解, 有的书籍甚至会省略推导过程。

但是, 了解这个过程很有益处, 因为如果读者了解这个过程, 就可以自己从头写一个深度学习框架。而且如果读者有志于进入 AI 行业, 在面试时也很可能被要求推导这里的过程。因此, 笔者在此会写明其中的详细过程, 希望读者能看到 BP 就是求导的过程, 并没有神秘之处。

值得一提的是, 初学者可能会混淆 BP 和 SGD 这两个概念。它们是不同的概念, BP 负责计算梯度 $\frac{\partial \text{LOSS}}{\partial w}$, SGD 负责从梯度训练网络。神经网络的训练是结合 BP 和 SGD。

回想此前的公式, 对于神经网络中的任何参数 w , 采用 SGD 的训练公式都是:

$$w^{\text{new}} = w - \eta \cdot \frac{\partial \text{LOSS}}{\partial w}$$

但如果 w 是网络中的隐神经元的参数，就看似难以直接计算偏导数 $\frac{\partial \text{LOSS}}{\partial w}$ 。解决方法是使用求导的链式法则，即，若 A 是关于 C_i 的函数，则：

$$\frac{\partial A}{\partial B} = \sum_i \frac{\partial A}{\partial C_i} \cdot \frac{\partial C_i}{\partial B}$$

如果不断运用链式法则，就会看到 BP 的过程，后文我们会计算一个更明显的例子。在此我们先以此前的简单网络为例。首先，由链式法则知：

$$\frac{\partial \text{LOSS}}{\partial w} = \frac{\partial \text{LOSS}}{\partial \text{OUT}} \cdot \frac{\partial \text{OUT}}{\partial w}$$

而对于 $\text{LOSS} = (\text{OUT} - \text{期望输出})^2$ ，有：

$$\frac{\partial \text{LOSS}}{\partial \text{OUT}} = 2 \cdot (\text{OUT} - \text{期望输出})$$

与 w 无关。因此我们的主要任务是计算 $\frac{\partial \text{OUT}}{\partial w}$ 。

考虑此前的网络，如图 2-25 所示，假设第 1 和第 2 层的非线性激活分别使用 f 和 g 。

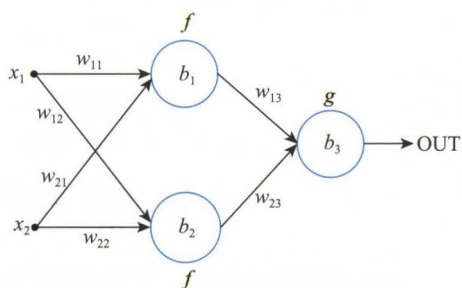


图 2-25 带非线性的神经网络

那么，输出公式是：

$$\text{OUT} = g(f(x_1 w_{11} + x_2 w_{21} + b_1) \cdot w_{13} + f(x_1 w_{12} + x_2 w_{22} + b_2) \cdot w_{23} + b_3)$$

不妨定义一些中间变量，例如用 I_k 代表第 k 个神经元的总输入，用 O_k 代表第 k 个神经元的输出（即 I_k 在非线性的激活后的值）：

$$I_1 = x_1 w_{11} + x_2 w_{21} + b_1, O_1 = f(I_1)$$

$$I_2 = x_1 w_{12} + x_2 w_{22} + b_2, O_2 = f(I_2)$$

$$I_3 = O_1 w_{13} + O_2 w_{23} + b_3, \text{OUT} = O_3 = g(I_3)$$

则可直接求导，例如：

$$\frac{\partial O_3}{\partial b_3} = \frac{\partial O_3}{\partial I_3} \cdot \frac{\partial I_3}{\partial b_3} = g'(I_3)$$

$$\frac{\partial O_3}{\partial w_{13}} = \frac{\partial O_3}{\partial I_3} \cdot \frac{\partial I_3}{\partial w_{13}} = g'(I_3) \cdot O_1$$

$$\frac{\partial O_3}{\partial w_{23}} = \frac{\partial O_3}{\partial I_3} \cdot \frac{\partial I_3}{\partial w_{23}} = g'(I_3) \cdot O_2$$

$$\frac{\partial O_3}{\partial b_1} = \frac{\partial O_3}{\partial I_3} \cdot \frac{\partial I_3}{\partial O_1} \cdot \frac{\partial O_1}{\partial I_1} \cdot \frac{\partial I_1}{\partial b_1} = g'(I_3) \cdot w_{13} \cdot f'(I_1)$$

$$\frac{\partial O_3}{\partial w_{11}} = \frac{\partial O_3}{\partial I_3} \cdot \frac{\partial I_3}{\partial O_1} \cdot \frac{\partial O_1}{\partial I_1} \cdot \frac{\partial I_1}{\partial w_{11}} = g'(I_3) \cdot w_{13} \cdot f'(I_1) \cdot x_1$$

$$\begin{aligned}\frac{\partial O_3}{\partial w_{21}} &= \frac{\partial O_3}{\partial I_3} \cdot \frac{\partial I_3}{\partial O_1} \cdot \frac{\partial O_1}{\partial I_1} \cdot \frac{\partial I_1}{\partial w_{21}} = g'(I_3) \cdot w_{13} \cdot f'(I_1) \cdot x_2 \\ \frac{\partial O_3}{\partial b_2} &= \frac{\partial O_3}{\partial I_3} \cdot \frac{\partial I_3}{\partial O_2} \cdot \frac{\partial O_2}{\partial I_2} \cdot \frac{\partial I_2}{\partial b_2} = g'(I_3) \cdot w_{23} \cdot f'(I_2) \\ \frac{\partial O_3}{\partial w_{12}} &= \frac{\partial O_3}{\partial I_3} \cdot \frac{\partial I_3}{\partial O_2} \cdot \frac{\partial O_2}{\partial I_2} \cdot \frac{\partial I_2}{\partial w_{12}} = g'(I_3) \cdot w_{23} \cdot f'(I_2) \cdot x_1 \\ \frac{\partial O_3}{\partial w_{22}} &= \frac{\partial O_3}{\partial I_3} \cdot \frac{\partial I_3}{\partial O_2} \cdot \frac{\partial O_2}{\partial I_2} \cdot \frac{\partial I_2}{\partial w_{22}} = g'(I_3) \cdot w_{23} \cdot f'(I_2) \cdot x_2\end{aligned}$$

然而这仍然烦琐，于是可再定义几个中间变量（梯度）：

$$\begin{aligned}G_3 &= \frac{\partial O_3}{\partial I_3} = g'(I_3) \\ G_2 &= \frac{\partial O_3}{\partial I_2} = \frac{\partial O_3}{\partial I_3} \cdot \frac{\partial I_3}{\partial O_2} \cdot \frac{\partial O_2}{\partial I_2} = G_3 \cdot w_{23} \cdot f'(I_2) \\ G_1 &= \frac{\partial O_3}{\partial I_1} = \frac{\partial O_3}{\partial I_3} \cdot \frac{\partial I_3}{\partial O_1} \cdot \frac{\partial O_1}{\partial I_1} = G_3 \cdot w_{13} \cdot f'(I_1)\end{aligned}$$

我们将在后文看到，它们的构造实际来自于 BP 过程。此时：

$$\begin{aligned}\frac{\partial O_3}{\partial b_3} &= \frac{\partial O_3}{\partial I_3} \cdot \frac{\partial I_3}{\partial b_3} = G_3 \\ \frac{\partial O_3}{\partial w_{13}} &= \frac{\partial O_3}{\partial I_3} \cdot \frac{\partial I_3}{\partial w_{13}} = G_3 \cdot O_1 \\ \frac{\partial O_3}{\partial w_{23}} &= \frac{\partial O_3}{\partial I_3} \cdot \frac{\partial I_3}{\partial w_{23}} = G_3 \cdot O_2 \\ \frac{\partial O_3}{\partial b_1} &= \frac{\partial O_3}{\partial I_1} \cdot \frac{\partial I_1}{\partial b_1} = G_1 \\ \frac{\partial O_3}{\partial w_{11}} &= \frac{\partial O_3}{\partial I_1} \cdot \frac{\partial I_1}{\partial w_{11}} = G_1 \cdot x_1 \\ \frac{\partial O_3}{\partial w_{21}} &= \frac{\partial O_3}{\partial I_1} \cdot \frac{\partial I_1}{\partial w_{21}} = G_1 \cdot x_2 \\ \frac{\partial O_3}{\partial b_2} &= \frac{\partial O_3}{\partial I_2} \cdot \frac{\partial I_2}{\partial b_2} = G_2 \\ \frac{\partial O_3}{\partial w_{12}} &= \frac{\partial O_3}{\partial I_2} \cdot \frac{\partial I_2}{\partial w_{12}} = G_2 \cdot x_1 \\ \frac{\partial O_3}{\partial w_{22}} &= \frac{\partial O_3}{\partial I_2} \cdot \frac{\partial I_2}{\partial w_{22}} = G_2 \cdot x_2\end{aligned}$$

可见，在计算出 $G_i = \frac{\partial \text{OUT}}{\partial I_i} = \frac{\partial O_3}{\partial I_i}$ 后，就容易算出 $\frac{\partial \text{OUT}}{\partial w_{ij}} = \frac{\partial O_3}{\partial w_{ij}}$ 。

最后乘上之前算出的 $\frac{\partial \text{LOSS}}{\partial \text{OUT}} = \frac{\partial \text{LOSS}}{\partial O_3}$ ，即可得到 $\frac{\partial \text{LOSS}}{\partial w_{ij}}$ ，可代入梯度下降公式，训练网络。

2.3.4 训练：实例

继续以此前的找大小网络为例，假设初始的网络的权重和偏置的情况如图 2-26 所示。

这里非线性激活的导数分别是：

□ 若 $x > 0$ ， $f'(x) = 1$ ，否则等于 0。

□ $g'(x)$ 恒等于 1。

若第 1 个训练样本为 $x_1 = 0.5$ ， $x_2 = 1$ ，则前向传播过程为：

$$I_1 = x_1 w_{11} + x_2 w_{21} + b_1 = 0.25, O_1 = \text{MAX}(0, I_1) = 0.25$$

$$I_2 = x_1 w_{12} + x_2 w_{22} + b_2 = 1.25, O_2 = \text{MAX}(0, I_2) = 1.25$$

$$I_3 = O_1 w_{13} + O_2 w_{23} + b_3 = 0.75, \text{OUT} = O_3 = I_3 = 0.75$$

$$\text{期望输出} = \text{MAX}(0.5, 1) = 1$$

使用上一节的公式，得到：

$$\frac{\partial \text{LOSS}}{\partial \text{OUT}} = 2 \cdot (\text{OUT} - \text{期望输出}) = -0.5$$

$$G_3 = g'(I_3) = 1$$

$$G_2 = G_3 \cdot w_{23} \cdot f'(I_2) = 0.5$$

$$G_1 = G_3 \cdot w_{13} \cdot f'(I_1) = 0.5$$

令学习速率 $\eta = 0.1$ ，再使用上一节的公式：

$$b_3^{\text{new}} = b_3 - \eta \cdot \frac{\partial \text{LOSS}}{\partial \text{OUT}} \cdot \frac{\partial \text{OUT}}{\partial b_3} = 0 - 0.1 \cdot (-0.5) \cdot G_3 = 0.05$$

$$w_{13}^{\text{new}} = w_{13} - \eta \cdot \frac{\partial \text{LOSS}}{\partial \text{OUT}} \cdot \frac{\partial \text{OUT}}{\partial w_{13}} = 0.5 - 0.1 \cdot (-0.5) \cdot G_3 \cdot O_1 = 0.5125$$

$$w_{23}^{\text{new}} = w_{23} - \eta \cdot \frac{\partial \text{LOSS}}{\partial \text{OUT}} \cdot \frac{\partial \text{OUT}}{\partial w_{23}} = 0.5 - 0.1 \cdot (-0.5) \cdot G_3 \cdot O_2 = 0.5625$$

$$b_1^{\text{new}} = b_1 - \eta \cdot \frac{\partial \text{LOSS}}{\partial \text{OUT}} \cdot \frac{\partial \text{OUT}}{\partial b_1} = 0 - 0.1 \cdot (-0.5) \cdot G_1 = 0.025$$

$$w_{11}^{\text{new}} = w_{11} - \eta \cdot \frac{\partial \text{LOSS}}{\partial \text{OUT}} \cdot \frac{\partial \text{OUT}}{\partial w_{11}} = 0.5 - 0.1 \cdot (-0.5) \cdot G_1 \cdot x_1 = 0.5125$$

$$w_{21}^{\text{new}} = w_{21} - \eta \cdot \frac{\partial \text{LOSS}}{\partial \text{OUT}} \cdot \frac{\partial \text{OUT}}{\partial w_{21}} = 0 - 0.1 \cdot (-0.5) \cdot G_1 \cdot x_2 = 0.025$$

$$b_2^{\text{new}} = b_2 - \eta \cdot \frac{\partial \text{LOSS}}{\partial \text{OUT}} \cdot \frac{\partial \text{OUT}}{\partial b_2} = 0 - 0.1 \cdot (-0.5) \cdot G_2 = 0.025$$

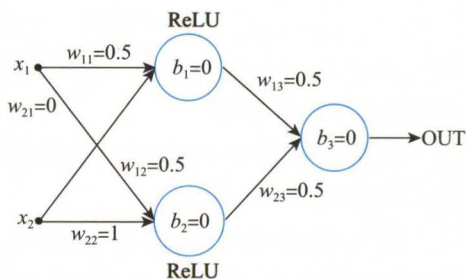


图 2-26 待训练的神经网络

$$w_{12}^{new} = w_{12} - \eta \cdot \frac{\partial \text{LOSS}}{\partial \text{OUT}} \cdot \frac{\partial \text{OUT}}{\partial w_{12}} = 0.5 - 0.1 \cdot (-0.5) \cdot G_2 \cdot x_1 = 0.5125$$
$$w_{22}^{new} = w_{22} - \eta \cdot \frac{\partial \text{LOSS}}{\partial \text{OUT}} \cdot \frac{\partial \text{OUT}}{\partial w_{22}} = 1 - 0.1 \cdot (-0.5) \cdot G_2 \cdot x_2 = 1.025$$

这就完成了 1 次训练。
实测网络经过 4000 组随机数据的训练后，参数如图 2-27 所示。

这与理论最优解已相当接近。因为 1.11×0.87 约等于 1， 0.79×1.23 约等于 1，其他参数都约等于 0，因此这个网络与此前的理论最优解是等价的。

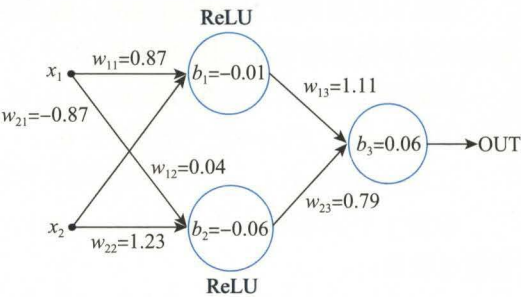


图 2-27 训练后的神经网络

2.3.5 训练：Excel 的实现

在 Excel 中的训练效果如图 2-28 所示。

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	eta	0.10																		
2																				
3	x	y	w11	w21	b1	w12	w22	b2	w13	w23	b3	I1	I2	OUT	目标	MSE	G	G3	G2	G1
4	0.50	1.00	0.5000	0.0000	0.0000	0.5000	1.0000	0.0000	0.5000	0.5000	0.0000	0.25	1.25	0.75	1.00	0.06250000	-0.05	1.00	0.50	0.50
5	0.91	0.42	0.5125	0.0250	0.0250	0.5125	1.0250	0.0250	0.5125	0.5625	0.0500	0.50	0.92	0.82	0.91	0.00744847	-0.02	1.00	0.56	0.51
6	0.94	0.74	0.5206	0.0287	0.0338	0.5213	1.0291	0.0347	0.5212	0.5784	0.0673	0.54	1.28	1.09	0.94	0.02373119	0.03	1.00	0.58	0.52
7	0.46	0.07	0.5055	0.0169	0.0178	0.5046	1.0159	0.0169	0.5044	0.5389	0.0365	0.25	0.32	0.34	0.46	0.01639928	-0.03	1.00	0.54	0.50
8	0.48	0.97	0.5115	0.0177	0.0307	0.5110	1.0168	0.0307	0.5109	0.5470	0.0621	0.29	1.27	0.91	0.97	0.00478426	-0.01	1.00	0.55	0.51
9	0.18	0.45	0.5149	0.0246	0.0378	0.5147	1.0242	0.0383	0.5150	0.5646	0.0759	0.14	0.59	0.48	0.45	0.00094507	0.01	1.00	0.56	0.51
10	0.41	0.27	0.5143	0.0232	0.0346	0.5141	1.0226	0.0348	0.5141	0.5609	0.0697	0.25	0.53	0.50	0.41	0.00668531	0.02	1.00	0.56	0.51

图 2-28 Excel 训练神经网络

首先，执行下列步骤：

- 1) 设置超参数：在 B1 填入学习速率。
- 2) 构造数据：在 A4 和 B4 填入 =RAND()。
- 3) 初始参数：在 C4 到 K4 填入初始参数。
- 4) 前向传播：设置 L4=A4*C4+B4*D4+E4，M4=A4*F4+B4*G4+H4，N4=MAX(0,L4)*I4+MAX(0,M4)*J4+K4。
- 5) 期望结果和损失：设置 O4=MAX(A4,B4)，P4=POWER(N4-O4,2)。
- 6) 反向传播，中间结果：设置 Q4=2*(N4-O4)*\$B\$1，R4=1，S4=R4*J4*IF(M4>0,1,0)，T4=R4*I4*IF(L4>0,1,0)（注，如果读者不熟悉 Excel，\$B\$1 的意思是在下拉时不自动更改单元格编号，因为这里固定学习速率在 B1 单元格）。
- 7) 反向传播，权重更新：设置 C5=C4-Q4*T4*A4，D5=D4-Q4*T4*B4，E5=E4-Q4*T4，F5=F4-Q4*S4*A4，G5=G4-Q4*S4*B4，H5=H4-Q4*S4，I5=I4-Q4*R4*MAX(0,L4)，J5=J4-Q4*R4*MAX(0,M4)，K5=K4-Q4*R4。

然后下拉 4000 行，即可看到最终训练出的权重。上节所述的结果就来自于此。

2.3.6 训练：反向传播

对于更复杂的多层网络，举例说明如何快速计算出 $\frac{\partial \text{OUT}}{\partial w_{ij}}$ ，并体验 BP 的运作过程，如图 2-29 所示。

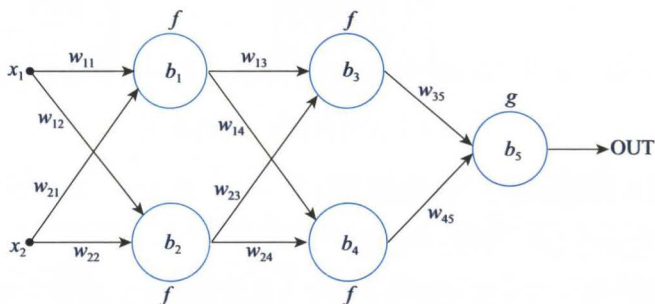


图 2-29 待训练的网络

令图中的 b_1 到 b_5 为 1 到 5 号神经元。如前所述，用 I_k 代表第 k 个神经元的总输入，用 O_k 代表第 k 个神经元的输出（即 I_k 在非线性激活后的值）。

可模仿此前的计算，先计算倒数第 1 层（5 号神经元）和倒数第 2 层（3 和 4 号神经元）的梯度：

$$G_5 = \frac{\partial O_5}{\partial I_5} = g'(I_5)$$

$$G_4 = \frac{\partial O_5}{\partial I_4} = \frac{\partial O_5}{\partial I_5} \cdot \frac{\partial I_5}{\partial O_4} \cdot \frac{\partial O_4}{\partial I_4} = G_5 \cdot w_{45} \cdot f'(I_4)$$

$$G_3 = \frac{\partial O_5}{\partial I_3} = \frac{\partial O_5}{\partial I_5} \cdot \frac{\partial I_5}{\partial O_3} \cdot \frac{\partial O_3}{\partial I_3} = G_5 \cdot w_{35} \cdot f'(I_3)$$

倒数第 3 层（1 和 2 号神经元）的计算很重要。注意从 2 号神经元到倒数第 2 层有 2 种路径，可以经过 4 号神经元，也可以经过 3 号神经元。同理，从 1 号神经元到倒数第 2 层也有 2 种路径。

这里的关键是，根据链式法则，在求导时需考虑所有可能的路径，将它们求和：

$$G_2 = \frac{\partial O_5}{\partial I_2} = \left(\frac{\partial O_5}{\partial I_4} \cdot \frac{\partial I_4}{\partial O_2} + \frac{\partial O_5}{\partial I_3} \cdot \frac{\partial I_3}{\partial O_2} \right) \cdot \frac{\partial O_2}{\partial I_2} = (G_4 \cdot w_{24} + G_3 \cdot w_{23}) \cdot f'(I_2)$$

$$G_1 = \frac{\partial O_5}{\partial I_1} = \left(\frac{\partial O_5}{\partial I_4} \cdot \frac{\partial I_4}{\partial O_1} + \frac{\partial O_5}{\partial I_3} \cdot \frac{\partial I_3}{\partial O_1} \right) \cdot \frac{\partial O_1}{\partial I_1} = (G_4 \cdot w_{14} + G_3 \cdot w_{13}) \cdot f'(I_1)$$

换言之，在计算倒数第 n 层的神经元的导数时，需考虑它与倒数第 $n-1$ 层的所有连接，将相应的导数求和。这就是 BP 的精髓。

在计算出所有 $G_k = \frac{\partial \text{OUT}}{\partial I_k} = \frac{\partial O_5}{\partial I_k}$ 后，对于 $\frac{\partial \text{OUT}}{\partial w_{ij}} = \frac{\partial O_5}{\partial w_{ij}}$ 的计算就很简单，与此前一致。

例如：

$$\begin{aligned}\frac{\partial O_5}{\partial w_{11}} &= \frac{\partial O_5}{\partial I_1} \cdot \frac{\partial I_1}{\partial w_{11}} = G_1 \cdot x_1 \\ \frac{\partial O_5}{\partial w_{23}} &= \frac{\partial O_5}{\partial I_3} \cdot \frac{\partial I_3}{\partial w_{23}} = G_3 \cdot O_2\end{aligned}$$

以此类推。

当批大小大于 1 时，也可以将 BP 写成矩阵的形式。在一定程度上，批大小越大，训练速度越快。

2.3.7 重要知识：梯度消失，梯度爆炸

仔细观察反向传播的过程，我们会发现：

- G_k 有可能在传播过程中绝对值越来越小（直到变成 0），这称为梯度消失（gradient vanishing），其会使得网络的训练停滞不前。
- G_k 有可能在传播过程中绝对值越来越大（直到发散），这称为梯度爆炸（gradient explosion），其会使得网络不稳定，性能崩溃。

举个梯度消失的例子。如果采用 sigmoid 或 tanh 非线性，在输入的绝对值很大的时候，会出现“饱和”（saturation），即导数趋近 0，根据 G_k 的公式，会造成梯度消失。

采用 ReLU 非线性可避免这个情况，因为 ReLU 不会饱和，在激活的时候（即输入为正数的时候）导数恒定为 1。

另一方面，ReLU 在输入为负数的时候，导数为 0，这称为“死 ReLU”（dying ReLU），往往是在学习速率过大时出现。这可通过使用 PReLU、RReLU 等变种解决，简单地说，就是让 ReLU 在输入为负数的时候也具有少量的导数。

再看梯度爆炸的例子。如果网络中的 w 很大，例如初始化网络时使用了过大的初始值，或是网络的权重随着训练越来越大，就可能发生梯度爆炸。对于循环神经网络（RNN）和 GAN，较为容易出现这种现象。

因此，如果发现网络的训练性能很差，值得做的事情就是观察网络内部梯度的流动情况，看是否出现了梯度消失和梯度爆炸。后文会说明如何在 MXNet 中查看网络内部的梯度情况。

改善梯度可通过多种技巧，包括后文将介绍的批规范化（BN）、残差网络（ResNet），还可采用梯度截断（gradient clipping）技术，即人工将过大的梯度减少，或引入一定的梯度惩罚（gradient penalty）。

2.3.8 从几何观点理解神经网络

如果读者的数学和空间想象力好，在此可用几何的观点理解神经网络。如果读者不熟悉这里的内容，可跳过本节。

推荐有能力的读者阅读 <http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/>，在此说明其中要点，如图 2-30 所示。

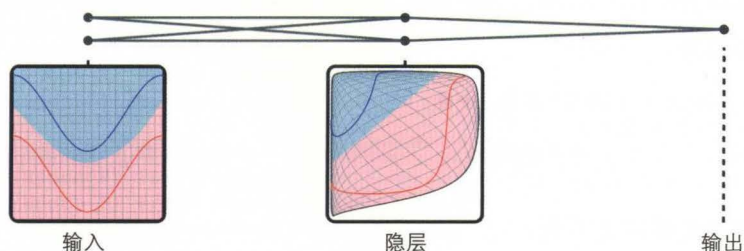


图 2-30 神经网络的几何理解

先看图中的左部区域。令蓝色和红色线代表需分开的数据，它们处于样本空间中。我们希望在样本空间找到图 2-30 所示的蓝色和红色区域，将蓝色线和红色线分开。但左图的区域分界是较为复杂的曲线，因此这个任务并不容易。

再看图 2-30 中所示的中部区域。经过神经网络隐层变换，相当于将样本空间不断变形（也可改变样本空间的维数），最终可让蓝色和红色区域的分界变得更简单，可直接用一条直线划分。这个思想也有些类似于 SVM 中的核方法。

对于更复杂的样本数据，划分的任务会更复杂，例如，有可能如图 2-31 所示。

如何将这里的蓝色和红色分开？根据拓扑学定理，所有 n 维流形都可以在 $2n+2$ 维空间中划分开。例如，图中的蓝色和红色线，在数学上看是 1 维流形，虽然在 3 维空间中无法解开，但可在 $2 \times 1 + 2 = 4$ 维空间中解开。

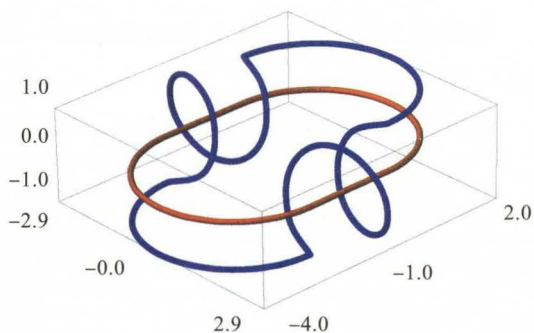


图 2-31 复杂的划分任务

神经网络在隐层使用大量神经元，就是在做升维，以便划分样本。这称为 disentangling，即将纠缠在一起的特征或概念分开。

2.3.9 训练：MXNet 的实现

下面我们将在 MXNet 训练一个“找大小”网络。我们在此前的神经元训练程序基础上，做少量修改。最开始的部分不变：

```
#-*-coding:utf-8-*- # 如果使用py文件，需声明此文件为 UTF-8 格式，这样可以使用中文注释。如
                        果使用ipynb文件，则无须此行
```

```
import logging
import math
import random
```

```
import mxnet as mx # 导入 MXNet 库
import numpy as np # 导入 NumPy 库, 这是 Python 常用的科学计算库

logging.getLogger().setLevel(logging.DEBUG) # 打开调试信息的显示
```

调整训练参数为:

```
n_sample = 10000 # 训练用的数据点个数
batch_size = 10 # 批大小
learning_rate = 0.1 # 学习速率
n_epoch = 10 # 训练 epoch 数
```

数据生成和迭代器定义不变:

```
# 每组数据是在 (0,1) 之间的 2 个随机数
train_in = [[ random.uniform(0, 1) for c in range(2)] for n in range(n_sample)]

train_out = [0 for n in range(n_sample)] # 期望输出, 先初始化为 0

for i in range(n_sample):
    # 每组数据的期望输出是 2 个输入数中的大者
    train_out[i] = max(train_in[i][0], train_in[i][1])

train_iter = mx.io.NDArrayIter(data = np.array(train_in), label = {'reg_
label':np.array(train_out)}, batch_size = batch_size, shuffle = True)
```

使用更大的网络 (后文会解释原因), 有 2 个隐层, 每个隐层有 10 个神经元, 使用 ReLU 非线性:

```
src = mx.sym.Variable('data') # 输入层
fc1 = mx.sym.FullyConnected(data = src, num_hidden = 10, name = 'fc1') # 全连接层
act1 = mx.sym.Activation(data = fc1, act_type = "relu", name = 'act1') # ReLU层
fc2 = mx.sym.FullyConnected(data = act1, num_hidden = 10, name = 'fc2') # 全连接层
act2 = mx.sym.Activation(data = fc2, act_type = "relu", name = 'act2') # ReLU层
fc3 = mx.sym.FullyConnected(data = act2, num_hidden = 1, name = 'fc3') # 全连接层
net = mx.sym.LinearRegressionOutput(data = fc3, name = 'reg') # 输出层
# 完成网络定义
module = mx.mod.Module(symbol = net, label_names = (['reg_label']))
```

训练语句为:

```
module.fit(
    train_iter, # 训练数据的迭代器
    eval_data = None, # 在此只训练, 不使用测试数据
    eval_metric = mx.metric.create('mse'), # 输出 MSE 损失信息
    # 将权重和偏置初始化为在 [-0.5, 0.5] 间均匀的随机数
    initializer = mx.initializer.Uniform(0.5),
    optimizer = 'sgd', # 梯度下降算法为 SGD
    # 设置学习速率
    optimizer_params = {'learning_rate': learning_rate},
    num_epoch = n_epoch, # 训练 epoch 数
    batch_end_callback = None, # 减少输出信息
    epoch_end_callback = None # 减少输出信息
)
```

训练输出的样例如下:


```
INFO:root:Epoch[0] Train-mse=0.014321
INFO:root:Epoch[0] Time cost=0.599
INFO:root:Epoch[1] Train-mse=0.000797
INFO:root:Epoch[1] Time cost=0.610
INFO:root:Epoch[2] Train-mse=0.000054
INFO:root:Epoch[2] Time cost=0.601
INFO:root:Epoch[3] Train-mse=0.000023
INFO:root:Epoch[3] Time cost=0.597
.....
INFO:root:Epoch[8] Train-mse=0.000010
INFO:root:Epoch[8] Time cost=0.641
INFO:root:Epoch[9] Train-mse=0.000008
INFO:root:Epoch[9] Time cost=0.663
```

可见最后 MSE 值非常低，说明网络训练成功。

可将网络的最终参数打印出来，方法为在程序最后加上：

```
for k in module.get_params(): # 对于所有参数...
    print(k) # 输出参数
```

网络的参数很多，输出样例如下：

```
{'fc2_weight':
[[ 0.0486978  0.08766525  0.08077119  0.33784643  0.08905049  0.29645106
  0.00288338  0.34725171 -0.07634521  0.10928501]
 [ 0.13954572 -0.11565644 -0.12410575 -0.20246539  0.39177299 -0.44348243
  0.44027492 -0.22734371 -0.11655849 -0.02233487]
 [ 0.26465911  0.30982044 -0.07922273 -0.02013446  0.06794196 -0.11245469
  0.38925287  0.33607876 -0.42896396 -0.16268948]
 [-0.42919746  0.14923854 -0.31610006 -0.1233644  0.37646568  0.56566089
  0.35641769 -0.35964924  0.37001216  0.41776562]
 [ 0.64992547 -0.02267127  0.54148632  0.31948808  0.06277914  0.25541821
  0.36146048  0.17887956 -0.38172558  0.32832971]
 [ 0.14103572  0.08160976 -0.38332352  0.03415999  0.43153968  0.23187535
  0.00922418 -0.39409238 -0.08533806 -0.03987403]
 [-0.29105067 -0.31619266  0.29837728  0.23691681 -0.03452195 -0.27323589
  0.1029952 -0.36478183 -0.4812102 -0.16728066]
 [ 0.26692805 -0.34508005  0.40280971 -0.25938466  0.20983706  0.11829288
  0.54039502  0.4025985  0.18182027  0.04902111]
 [-0.1484715  0.11306345 -0.01297597  0.40086839  0.19358653 -0.41355854
 -0.4167673  0.46980906  0.1667667  0.14709415]
 [ 0.16971977 -0.32909042 -0.28967363 -0.14184782 -0.37113082  0.25068617
 -0.18457165  0.10783064 -0.13628924 -0.17498057]]
<NDArray 10x10 @cpu(0)>, 'fc1_weight':
[[ 0.47274888 -0.52280331]
 [-0.08552073  0.07267038]
 [ 0.6828323  0.45960492]
 [-0.40226099  0.09994139]
 [-0.28393939  0.26388767]
 [-0.48189631  0.50735527]
 [ 0.31580904  0.25287443]
 [-0.24670839 -0.0856314 ]
 [-0.03368923 -0.0253025 ]
 [-0.29325953  0.29024765]]
<NDArray 10x2 @cpu(0)>, 'fc3_bias':
[ 0.10837685]
<NDArray 1 @cpu(0)>, 'fc3_weight':
[[-0.28320521 -0.15519413 -0.37932032  0.38324216  0.78082341 -0.1781657
```

```

-0.41277334  0.62705952 -0.35916656  0.44948524]]
<NDArray 1x10 @cpu(0)>, 'fc2_bias':
[-0.24409188 -0.09624054 -0.18390927  0.07458802 -0.02778439 -0.05218391
 0.02712426 -0.01276529  0.22903292 -0.0126054 ]
<NDArray 10 @cpu(0)>, 'fc1_bias':
[ 0.03458993 -0.07169167 -0.02335646 -0.06092598  0.01711669 -0.00467092
-0.01076672  0.          0.          0.00424089]
<NDArray 10 @cpu(0)>}

```

这里的 NDArray 是 MXNet 内部的一种数组类型，它的优势是同时支持 CPU 和 GPU 加速计算，也可与 Numpy 数组相互转换。

如果将中间层的神经元个数减少，例如将 10 减为 3，我们将发现网络容易性能停滞，MSE 无法降低，这称为欠拟合，实际是因为陷入了局部极值：

```

INFO:root:Epoch[0] Train-mse=0.023845
INFO:root:Epoch[0] Time cost=0.592
INFO:root:Epoch[1] Train-mse=0.014110
INFO:root:Epoch[1] Time cost=0.594
INFO:root:Epoch[2] Train-mse=0.014104
INFO:root:Epoch[2] Time cost=0.595
INFO:root:Epoch[3] Train-mse=0.014102
INFO:root:Epoch[3] Time cost=0.591
.....
INFO:root:Epoch[8] Train-mse=0.014092
INFO:root:Epoch[8] Time cost=0.595
INFO:root:Epoch[9] Train-mse=0.014090
INFO:root:Epoch[9] Time cost=0.591

```

换言之，使用更复杂的网络、更多的神经元，更容易得到更好的结果。这是深度学习的一个原则。

修改权重的初始化参数，对于训练性能也会有较大影响。例如，如果去掉代码中指定的初始化方法，采用 MXNet 默认的初始化方法，网络的训练同样容易停滞：

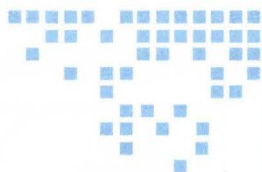
```

INFO:root:Epoch[0] Train-mse=0.057610
INFO:root:Epoch[0] Time cost=0.595
INFO:root:Epoch[1] Train-mse=0.055209
INFO:root:Epoch[1] Time cost=0.603
INFO:root:Epoch[2] Train-mse=0.055209
INFO:root:Epoch[2] Time cost=0.596
INFO:root:Epoch[3] Train-mse=0.055209
INFO:root:Epoch[3] Time cost=0.597
.....
INFO:root:Epoch[8] Train-mse=0.055209
INFO:root:Epoch[8] Time cost=0.601
INFO:root:Epoch[9] Train-mse=0.055209
INFO:root:Epoch[9] Time cost=0.594

```

可见，网络的训练有时确实需要做不少调整，这就是调参的微妙之处。

业内著名的调侃说法是“深度学习就是调参”。当然，这只是开玩笑。如果使用更先进的网络架构，尤其是加入后文会介绍的 BN 层后，可将网络对于调参的敏感性大大降低。



深度卷积网络：第二课

3.1 重要理论知识

在此我们补充重要的理论知识。从数学上看，神经网络的训练是一个优化问题，其中会涉及许多机器学习中的概念。

3.1.1 数据：训练集、验证集、测试集

机器学习的第一步是收集数据。不过，当我们拥有大量数据后，并不会直接将它们全部用于训练模型。这是因为我们训练网络的目标是希望它既能在现有数据上取得较佳性能，又能在未来的新数据（out-of-sample data）上取得较佳性能。就像训练 AlphaGo 的策略网络时，我们希望它能学会棋谱的下法，同时在面对棋谱里没有的全新的局面时也能给出较好的下法。

所以训练过程应该分为两部分：

- 保证能学好现有的数据。这往往需要调整一些超参数（hyper-parameters）。如果学不好，称为欠拟合，后文会进一步分析。
- 检验在新数据上的性能。这称为泛化能力。
 - 如果在旧数据上性能好，在新数据上性能差，说明泛化能力差，称为过拟合。
 - 如果在新数据上性能也好，称为泛化能力强，是我们追求的目标。

然而如果我们只有固定的一批数据（例如固定的几十万局棋谱），怎样检验网络在新数据上的性能？

答案是，我们可以将数据分成两部分，并始终只用其中一部分训练网络，于是剩下的数据对于网络就始终是全新的，可用于检验网络在新数据上的性能。

最严格的做法是将数据分为三部分：其中训练集用于训练网络，验证集用于调整网络的超参数，测试集用于检测和预测网络对于未来的新数据的效果，如图 3-1 所示。

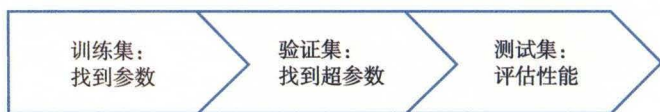


图 3-1 训练集，验证集，测试集

对于传统的机器学习方法，这三部分的数据比例可取 80%、10%、10%。不过对于深度学习，由于训练数据量往往很大，我们可增大训练集的比例，例如取 90%、5%、5%，甚至更悬殊。

典型的训练过程是：

- 1) 选定一组超参数，用训练集训练网络。
- 2) 调整超参数，直到可在训练集上获得较好效果。
- 3) 在验证集上运行网络，看效果如何。注意，只是运行网络，不能用验证集训练网络。
- 4) 调整超参数，直到可在训练集和验证集上都获得较好效果。注意，虽然我们没有用验证集训练网络，但由于超参数调整过程考虑到了验证集，因此严格说来，不能认为验证集的性能就能代表网络在全新数据上的性能。
- 5) 最后，对于超参数满意后，在测试集上运行网络，得到的效果就代表了网络在全新数据上的性能。注意，不能再根据网络在测试集的效果再修改超参数（虽然没有人能阻止你这样做），因为如果再去调整，那就和验证集没有区别了，也就失去了区分验证集和训练集的意义。

上述过程可谓完善。但令人遗憾的是，很多时候我们还是会根据测试集调整超参数。譬如，如果我们发现网络最终在测试集上的效果很差，那无疑会想方设法再继续实验，不可能就此放弃。

因此，许多研究中已不使用验证集。那么，我们可选取数据的 90% 作为训练集，10% 作为测试集。如果数据特别多，还可进一步增大训练集的比例。

最后，如果永远选取数据中固定的一部分作为测试集，严格说仍然不够有说服力。为此，可使用交叉验证（cross-validation）方法，即：

- 1) 将数据分为 k 份。
- 2) 初始化网络。选取数据中的第 1 份作为测试集，其余作为训练集，得到一个结果。
- 3) 重新初始化网络。选取数据中的第 2 份作为测试集，其余作为训练集，得到一个结果。
- 4) ……
- 5) 重新初始化网络。选取数据中的第 k 份作为测试集，其余作为训练集，得到一个结果。

6) 对所有结果取平均，即为一个更有说服力的结果。

不过，由于深度网络的训练往往较慢，因此交叉验证过程会很慢，实际在训练深度网络时使用交叉验证方法的人不多。

3.1.2 训练：典型过程

假设我们的数据共有 10000 个（即 10000 个输入和 10000 个期望输出），批大小为 100。那么训练网络的典型过程如下：

- 1) 读入所有数据。
- 2) 将数据划分为训练集和测试集，例如前 9000 个作为训练集，后 1000 个作为测试集。将这个划分固定下来。
- 3) 开始第 1 个 epoch。将训练集的 9000 个数据打乱，注意输入和输出保持正确的对应。
- 4) 将第 1 个 batch，即打乱后的数据的第 1~100 号，送入网络。
- 5) 正向传播。将结果与期望值比较，计算损失。
- 6) 反向传播。更新网络参数。
- 7) 将第 2 个 batch，即打乱后的数据的第 101-200 号，送入网络。重复上述过程。
- 8) 在所有训练集完成后，即经过 $9000/100=90$ 个 batch 后，完成第 1 个 epoch。
- 9) 将测试集送入网络。正向传播。将结果与期望值比较，计算损失。
- 10) 观察训练集的损失和测试集的损失，做一些事情，例如，可能需调整学习速率，或停止训练，或停止训练并修改超参数。
- 11) 开始第 2 个 epoch。将训练集的 9000 个数据再次打乱。重复上述过程。

3.1.3 有监督学习：回归、分类、标签、排序、Seq2Seq

在机器学习中，常见的是有监督学习（supervised learning）。即在数据集中，对于每个输入 X ，都有现成的输出目标 Y （可称为标签），模型的目标是从 X 找到 Y 。

在有监督学习中，最常见的问题是回归（regression）和分类（classification）。

回归的目标是从输入 X 给出一个数字 Y ，例如：

- 从一张人物照片，给出其中人物的年龄。
- 从一段股票走势，给出明天的涨跌幅。
- 从一张户外照片（见图 3-2），给出下雨的概率。
- AlphaGo 的价值网络：从当前棋盘局面，给出胜率。

这里的关键是，数字 Y 往往是连续的，而且越靠近真实值越好。

例如，如果照片中人物的真实年龄是 23.5 岁，那么给出 22.7 岁就比给出 39.8 岁要更好。所以损失函数往往会使用 MSE 损失。



图 3-2 回归问题的实例

分类的目标是从输入 X 给出它属于哪一个类别 Y ，例如：

- ❑ 从一张明星照片，给出其中的明星的名字。
- ❑ 从一张包含一个字母的图像（见图 3-3），给出其中的字母是多少。
- ❑ 从一段音乐，给出其中的歌名。
- ❑ AlphaGo 的策略网络：从当前棋盘局面，给出下一手的位置。



图 3-3 分类问题的实例

这里的关键是，类别 Y 往往是离散的，而且目标是完全正确。

例如对于字母识别问题，有 A 到 Z 共 26 个类别。如果真实值为 E，那么只有给出 E 才是正确的。如果给出 D，或给出 A，或给出 Z，错误程度都一样。所以损失函数会使用后文将提到的交叉熵（cross-entropy）损失。

严格说来，这个做法有时值得商榷。例如，如果将图像中的哈士奇犬识别为阿拉斯加雪橇犬，虽然是错误的，但会比识别为美国短毛猫要更准确。不过，目前大家在训练模型时一般不会这样精细地处理。

在实际运行中，网络的目标是给出图像属于每一个类别的概率。例如对于字母识别的问题，会给出图像中的字母是 A 的概率、是 B 的概率、是 C 的概率等。根据概率学，所有概率总和应为 1，所以我们在输出时会使用后文会提到的 SoftMax 层。

需要注意的是，当回归的目标是给出单 1 个概率时，例如预测明天下雨的概率，就与分类问题相似，可认为是将明天分类为 2 种：“会下雨的明天”和“不会下雨的明天”。如果“会下雨的明天”的概率是 p ，那么“不会下雨的明天”的概率就是 $1-p$ 。

此时我们可用 sigmoid 作为输出的非线性激活函数，它可保证概率 p 在 0 和 1 之间，符合概率的定义。

此外，还有标签（tagging）问题，它和分类的区别是所有类别的概率相加可大于 1，因为一个样本可有多个标签。

还有排序（ranking）问题，它与回归很相似，譬如可先回归到分数，然后输出按分数排序的结果。排序与回归的区别是，排序不关注具体的分数，只关注最终的排序结果。

最后，还有 Seq2Seq（序列变换）问题，它试图将一段序列（长度可变）转化为另一段

序列（长度可变），例如：

- ❑ 翻译：将一段中文语句，转化为一段英文语句。
- ❑ 语音识别：将一段语音录音，转化为一段语句。
- ❑ 聊天机器人：将用户输入的语句生成电脑的回答语句。

Seq2Seq 问题往往会使用 RNN（循环神经网络），特别是 LSTM（长短期记忆模型）。

3.1.4 无监督学习：聚类、降维、自编码、生成模型、推荐

再看无监督学习（unsupervised learning）。无监督学习的特点是，只有大量数据 X ，没有对应的标签 Y ，例如：

聚类（clustering）：将数据集的样本自动划分为几簇（cluster），每簇内的样本有类似之处。例如，给定许多顾客的特征，自动将类似的顾客聚在一起，形成几簇，如图 3-4 所示。这是典型的无监督学习问题，即无须告诉模型每一类顾客的特征，模型会自行发现。

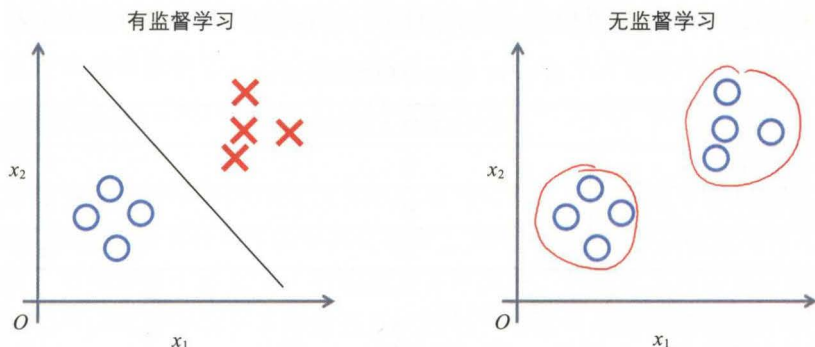


图 3-4 有监督学习与无监督学习

降维（dimensionality reduction）：这个概念较为抽象。在数学上看，样本的数据往往处于高维空间，这不利于后续处理，我们希望找到办法将其投射到低维空间并尽可能保留信息。很好的例子是自然语言处理中的 Word2Vec 方法。

生成模型（generative model），包括自编码（auto-encoding）和生成式对抗网络（GAN）。只需输入大量无标签的图像，模型就能自动生成类似的图像。

此外，常见的问题还有推荐（recommendation）。这是业界的热门问题，因为可以直接创造效益，如提升用户在页面的停留时间、提升用户的页面访问量、提升用户的购买金额。

例如，根据用户的阅读习惯，推荐用户可能感兴趣的新闻。还有购物推荐、APP 推荐、游戏推荐、电影推荐、音乐推荐、书籍推荐等。

推荐的常用实现方法有两种：

- ❑ 协同过滤：如果计算出用户 A 和用户 B 的口味相似，那么可向用户 A 推荐用户 B 喜欢的东西。
- ❑ 关联规则：如果计算出购买了 X 的用户往往会购买 Y，那么可向购买了 X 的用户

推荐 Y。

最后，还有半监督学习（semi-supervised learning），即数据中只有部分样本带有标签，然后希望给所有样本和未来的样本找到标签。

半监督学习很实用，因为标签往往是人工加上的，如果只加一部分标签，就能实现全部有标签的情况的准确率，那么可节省大量人力。

一种有趣的方法是，先人工标记少量标签，然后从少量标签训练网络，然后让网络预测所有样本的标签，再人工筛选和修改其中的标签，重复这个过程。由于网络的预测会越来越准，因此可节省许多人工标注的时间。

3.1.5 训练的障碍：欠拟合、过拟合

在训练过程中，两大障碍是欠拟合（under-fitting）和过拟合（over-fitting）。如前所述，数据会分为训练集和测试集。如果在训练集上性能差，称为欠拟合；如果在训练集上性能好，在测试集（或新数据）上性能差，称为过拟合。两者的关系和区分方法如表 3-1 所示。

表 3-1 过拟合和欠拟合对比

	在训练数据上的性能	在新数据上的性能
最佳情况：泛化能力高	好	好
过拟合：泛化能力低	好	差
欠拟合	差	差

读者可能会问：是否有可能在训练集上性能差，在测试集上性能好？对于某些训练方法，如使用数据增强、Dropout、RReLU 后，这有可能。但在正常情况下，这几乎不可能发生。

前文将数据划分为训练集、测试集，有助于及时发现过拟合，以便采取对策。正常情况下，我们会观察到，随着训练 epoch 的增长，模型在训练集和测试集的性能变化如图 3-5 所示。

右图中的红线是模型在训练集的性能，应随着训练越来越高。如模型在测试集的性能为绿线，那么越接近红线，就代表模型运作越良好。如模型在测试集的性能为蓝线，离红线的差距很大，甚至越训练越差，就说明出现了明显的过拟合。因此还有一种方法叫提前终止（early-stopping），即如果发现测试集的性能越训练越差，就应该停止训练。

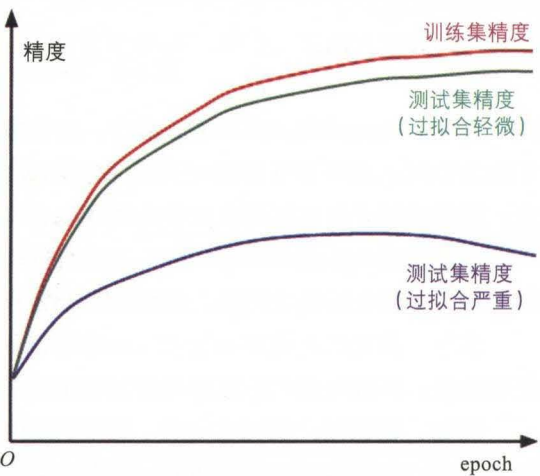


图 3-5 训练集精度与测试集精度随 epoch 的变化

出现欠拟合，通常有两种原因：

- 网络的容量 (capacity) 不足。这往往是因为网络的广度和深度不够。可通过增大网络的规模解决。
- 如果网络规模很大，仍然拟合效果差，那就是因为网络陷入了局部的极值，或者网络的拟合速度不够。
 - 这可能是由于训练的超参数不够妥当，如学习速率过大或过小，初始化方法不佳，或网络的架构不适合。可通过相应的调整解决。
 - 但也有可能是由于数据的问题，如数据本身的噪声过大、分布过于极端等。这时需要调整数据，并仔细设计网络的架构。

过拟合更为微妙。根据传统机器学习理论，一般认为参数过多、拟合能力过强的模型容易出现过拟合。例如，如果用高次多项式去拟合数据点，可以极好地拟合训练集中的每个点，但很可能并不符合数据背后的规律，无法泛化到新数据中。而且，实际的数据往往有噪声，如果我们简单地要求模型 100% 吻合数据，很可能会把噪声也记忆下来，影响模型的性能。

由于深度网络的参数极多，拟合能力很强，那么它是否就容易出现过拟合？我们可在网络中引入正则化 (regularization)，以试图找到更稳定的参数，提高它的健壮度，改善过拟合。这会在后文介绍。

不过，目前的深度网络和训练方法实际已相当健壮，尤其是深度卷积网络，它拥有一定的自我正则化能力，不容易出现简单的过拟合。如果出现过拟合，往往与训练数据有关。

简单地说，如果数据不够充分，不够覆盖各种情况，就容易误导网络。例如，对于图像识别问题，如果训练集中的苹果图像都是红色的，那么网络有可能出现“苹果必须是红色的”的错觉，在未来遇到一张青苹果的图像时，就有可能认为它不是苹果。

为此，一方面需要更大的训练集，另一方面可使用数据增强 (data augmentation) 技术，从现有的样本中生成更多的样本，改善网络的训练。我们会在后文介绍。

3.1.6 训练的细节：局部极值点、鞍点、梯度下降算法

从数学角度而言，梯度下降属于贪心算法，对于非凸 (non-convex) 的问题 (绝大多数问题是非凸的)，有可能陷入局部极小值点 (local minimum)，无法保证到达全局极小值点 (global minimum)。这就属于欠拟合。

如图 3-6 所示，从不同的起点开始，经过梯度下降，往往会最终到达不同的局部极小值点，无法保证最终到达的一定就是全局的极小值点。

不过，根据近年来的研究，如《The Loss Surfaces of Multilayer Networks》^①，对于大规模的神经网络，这实际影响不大。如果神经网络的规模够大够深，使用足够多的神经元，往往最后会得到相当靠近全局最优值的解。

① 地址为 <https://arxiv.org/abs/1412.0233>。

简单解释是，极小值点的特点是附近的所有维数都朝上走，但大规模神经网络的参数空间的维数太高，很少会出现附近所有维数都是朝上走的情况，因此在训练的早期很难遇到极小值点，而在遇到极小值点的时候，往往网络的性能已经较高了。此外，SGD 本身会引入一定的噪声，所以网络并不容易陷入局部极值点。

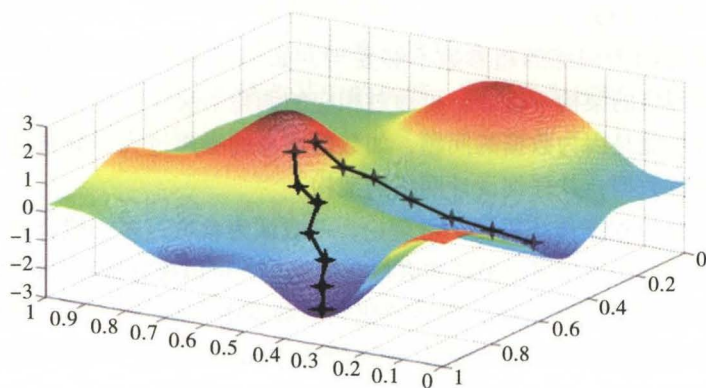


图 3-6 梯度下降

在实际训练过程中，会对梯度下降造成障碍的是鞍点 (saddle point)。如图 3-7 所示，鞍点附近的部分维数是朝上走的，部分维数是朝下走的。

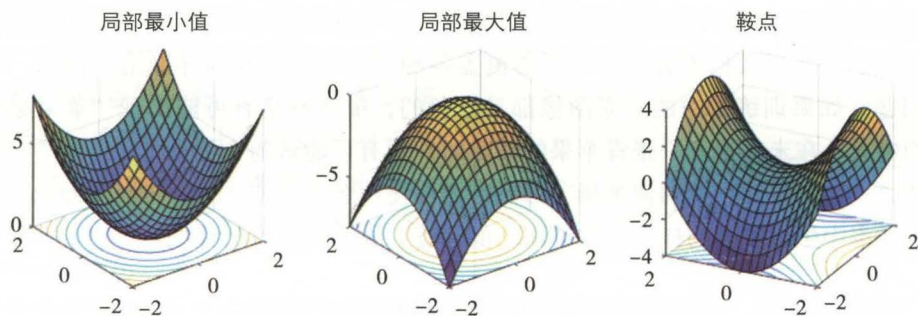


图 3-7 局部最小值，局部最大值，鞍点

由于鞍点仍然有维数是朝下走的，因此梯度下降法最终会脱离鞍点，但可能会在鞍点停滞一段时间才能脱离，影响训练速度。为此，业界发展出 momentum、Adagrad、Adam 等在 SGD 基础上改进的梯度下降算法，他们的训练速度往往更快。其中 momentum 的思想是典型的，它相当于把下山的过程变成一个有一定重量的铁球滚下山的过程，加入了惯性。

现有的深度学习框架都内置了这些算法，我们可在训练时实验出效果最好的选择。读者可访问 <http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html>，看到它们的动画演示。

机器学习中的著名定理是“没有免费午餐” (no free lunch theorem)，即不会有一种万能的永远比其他优化算法更好的优化算法，一切都需由实际实验决定。

根据经验，如果数据集很复杂，那么普通的 SGD 虽然速度更慢，但有可能会得到更好的准确率。例如 AlphaGo 的训练就是使用普通的 SGD。不过 AlphaGo Zero 的训练又加入了 momentum。

因此，如果读者需要训练出最佳效果，可实验多种梯度下降算法，都训练一段时间，然后选取其中实际效果最佳的算法，进一步训练。这也是我们实验超参数的通用方法。

3.2 神经网络的正则化

过拟合是提高神经网络性能的主要障碍，而神经网络的正则化（regularization）是为了避免过拟合。研究人员已发展出多种方法，本节会重点介绍。

3.2.1 修改损失函数：L2 和 L1 正则化

根据著名的奥卡姆剃刀（Occam's Razor）原则：

如果关于同一个问题有许多种理论，每一种都能做出同样准确的预言，那么应该挑选其中使用假定最少的；尽管越复杂的方法通常能做出越好的预言，但是在不考虑预言能力的情况下，前提假设越少越好。

于是，我们希望网络越简单越好：

□ 最简单的网络没有连接，因此我们希望网络的连接越少越好。

□ 如果连接的权重为 0 就相当于没有连接，因此我们希望网络中连接的权重越小越好。

这就是 L2 和 L1 正则化的基本思想。以上是粗略理解，严格说来，它们与数据和噪声的先验分布有关。

具体做法是在原有的损失函数上加上一个正则化项。最常用的是 L2 正则化：

$$\text{LOSS}_{\text{L2}} = \text{LOSS} + \lambda \sum w^2$$

其中 λ 为系数， Σ 是对所有权重求和。

观察上式，当最小化 LOSS_{L2} 时，网络就会试图让网络的所有权重 w 都靠近 0。如果 λ 越大，就说明越希望 w 靠近 0，但也越可能会影响网络的训练性能（因为部分 w 不应该靠近 0）。因此 λ 的值需要由实验决定，通常可先取一个很小的值，如 0.0001。

可具体计算此时的权重更新公式：

$$\begin{aligned} w^{\text{new}} &= w - \eta \cdot \frac{\partial \text{LOSS}_{\text{L2}}}{\partial w} = w - \eta \cdot \frac{\partial \text{LOSS}}{\partial w} - \eta \cdot \frac{\partial (\lambda w^2)}{\partial w} \\ &= w \cdot (1 - 2\eta\lambda) - \eta \cdot \frac{\partial \text{LOSS}}{\partial w} \end{aligned}$$

可见， λ 的实际作用就是让 w 在训练的每一步都乘上 $1-2\eta\lambda$ 因子，略为衰减。

此外常用的还有 L1 正则化：

$$\text{LOSS}_{\text{L1}} = \text{LOSS} + \lambda \sum |w|$$

有兴趣的读者可计算它对于权重更新公式的影响，计算并不复杂。

3.2.2 修改网络架构：Dropout 正则化

上述正则化方法是通过修改损失函数实现的。为了避免过拟合，还可通过修改网络架构实现。

例如 Dropout 正则化，来自 2012 年的深度神经网络 AlexNet。如图 3-8 所示，考虑左图所示的神经网络，在每次训练时，如右图所示，随机删除 50% 的隐神经元（将这些神经元的所有权重视为 0，并且在更新权重时不更新它们的权重）。

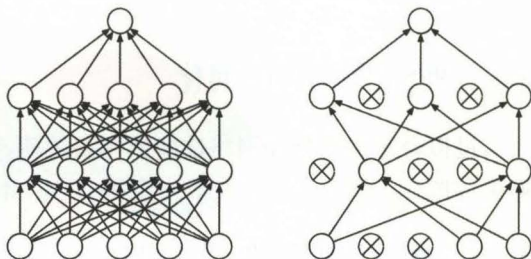


图 3-8 Dropout 正则化

然后恢复所有神经元。在下次训练时，再随机删除 50% 的隐神经元。由于每次训练会随机删除不同的神经元，这实际上会要求每个神经元都尽可能发挥自己的作用。

这里的思想是，如果用普通的训练方法，有可能网络中只有部分神经元在实际发挥作用，其余神经元在“围观”。但如果用 Dropout 训练方法，会尽可能让每个神经元都不能坐等其他神经元完成任务（因为其他神经元可能被删除掉），于是每个神经元都会更多地得到训练的刺激。

后文在讨论残差网络时还会提到两种采用类似思想的正则化方法：随机深度和 Shake-Shake 正则化。

通常我们会在神经元密集的层使用 Dropout 正则化。例如，从 1000 个神经元到 1000 个神经元的连接有 1000000 个，容易出现过拟合，用 Dropout 就可以改善这一点。具体删除神经元的比例可根据实验调整。

在 MXNet 中使用 Dropout 层的方法很方便。假设上一层为 net，可加入：

```
dropout = mx.sym.Dropout(net, p=0.3)
```

那么 dropout 就是随机删除 30% 后的结果。

3.2.3 更多技巧：集合、多任务学习、参数共享等

1. 集合

从另一个角度看，Dropout 训练方法就像训练了很多个只有 50% 隐神经元的网络，然后考虑它们的集合（ensemble），类似于对这些网络的输出结果做平均。

使用多个模型的集合，本身也是一种正则化方法，可提高网络的准确率。例如，我们可训练多次网络，由于每次的初始权重是随机初始化的，且 SGD 也带有随机性，因此最终会得到多个不同的网络。如果对这些网络的输出做平均，往往会比单一网络的输出更可靠。

我们还可训练多个采用不同架构的网络，让网络之间的差异更大，这样它们有可能会注意到数据的不同方面，在集合后，进一步减少过拟合。

集合方法的代价是运行速度更慢，因此研究人员通常仅在参加比赛（大家称为“刷榜”）时会使用集合，在实际运用中较少用到。

2. 多任务学习

我们可让网络同时完成多个相关的任务，那么网络需要找到同时满足各个任务的特征，因此特征的普适性更高。典型的例子是 AlphaGo Zero。

再举个多任务学习的例子。如果需训练一个预测图像中人物性别的网络，那么可尝试让网络同时预测图像中人物的性别和年龄。这里有一些细节：

- ❑ 数据集中需包括图像中人物性别和年龄的信息。
- ❑ 需选择合理的损失函数，例如可定义性别预测的损失是 { 正确：0 分，错误：100 分 }，年龄预测的损失是年份误差的平方，然后将 2 个损失相加作为最终的损失。

3. 参数共享

多任务学习，可认为是将多个实现不同任务的网络共享参数。

后文的卷积网络也相当于在不同神经元间共享参数，因为它会将同一个卷积核施加于图像的各个区域上。

因此，卷积网络本身就具有一定的自我正则化能力，这是它为什么如此有效的原因之一。由于后文介绍的批规范化（BN）层也有一定的正则化能力，所以在使用了 BN 层的深度卷积网络中，不一定再需要额外的正则化。

4. 标签平滑 (label smoothing) 与噪声标签 (noise label)

在分类问题中，研究人员常将图像的分类目标称为图像的标签 (label)。

如果训练数据中某张图像属于某一类别，那么简单的想法是设置这张图像属于这一类别的概率为 100%，然后希望网络尽量朝着这一目标趋近。

但经过研究人员分析发现，这可能会造成网络中的权重越来越大（这来自于 SoftMax 层的特性，它需要输入无穷大才能输出 100%），不利于网络的稳定性。而且，训练数据中的标签也不一定完全正确和全面，因为这些标签往往是人工加上的，有时会有错漏。

因此，我们可尝试将概率设置为更小的数（例如 95%），这称为标签平滑。也可在一定概率下将标签改变，这称为噪声标签。读者可参考 MXNet 中 SoftmaxOutput 的 smooth_alpha 参数。

与之相关的正则化方法是 mixup^①。令 x 为样本， y 为标签，我们可从训练数据 (x_i, y_i) 和 (x_j, y_j) 中创造出虚拟训练样本，用于增强数据集：

$$(\lambda x_i + (1-\lambda)x_j, \lambda y_i + (1-\lambda)y_j)$$

其中 λ 是随机数，线性混合方法是简单的点对点混合。

^① <https://arxiv.org/abs/1710.09412>。

5. 提前终止 (early-stopping)

如果发现测试集的性能越训练越差，就应该停止训练。这是一个简单而有效的正则化方法，还可节约训练时间。

3.2.4 数据增强与预处理

数据对于深度学习尤为重要。研究人员发现，最关键的改善过拟合的方法，往往是提高数据的质量。数据越丰富，训练效果越好，而且可通过数据增强 (data augmentation) 进一步提高数据的丰富性。

举例，如果网络能很好地识别训练集中的图像 X，但如果将 X 移动、旋转、放大或缩小一些，网络就识别不好，这就属于过拟合。

如果按照传统的机器学习做法，应该构建一个“对移动、旋转、放大或缩小拥有天生的免疫能力”的网络。这是一个困难的任务。目前的深度卷积神经网络架构可以在一定程度上做到这一点，但并不能保证。

目前流行的做法是，不再去想如何找到这种网络，而是在训练时就将图像 X 做各种各样的移动、旋转、放大或缩小等处理，将得到的图像 X' 也同时送入训练；这就是数据增强。每次让网络看到不同的训练样本，可提高网络对于各种情况的适应能力，效果显著。

对于图像而言，可在每次训练前将图像做镜像、平移、剪切、旋转、放缩、变形、改变亮度和对比度、色彩调整、加入噪声、将随机区域替换为噪声等。随着 GAN 的发展，还可通过 GAN 生成更多的数据并用于训练。

常用的图像增强库包括 <https://github.com/mdbloice/Augmentor>、<https://github.com/aleju/imgaug> 等。生成图像的效果如图 3-9 所示，其中最左上角的图是原图，其他是对原图做各种变换后得到的图像。

由图可见，适度的图像增强可让网络看到更丰富的样本。同时，图像增强的幅度也不能太大，如果变换后的图像连人类都难以识别，那就有可能造成网络的欠拟合。

再以围棋的情况举例。注意到棋盘局面通过镜像和旋转，总共可生成 8 种等价的局面：原始局面、旋转 90 度、旋转 180 度、旋转 270 度、镜像、旋转 90 度并镜像、旋转 180 度并镜像、旋转 270 度并镜像。

因此我们在每次训练时，会将每个局面在 8 种变换方法中随机选一种（如果是训练策略网络，记得相应地修改下一手的位置）。这就相当于将训练的棋谱数提高到了原来的 8 倍，可明显地减少过拟合。

在运行网络时，我们也可对样本做各种变换，将变换后的图像输入网络，然后取网络输出的平均值，这往往可以获得更准确的预测值，代价是运行速度更慢。

数据预处理对于网络的性能也有一定影响。举例，图像文件的原始像素往往位于 [0, 255] 区间，如果直接将这样大的数字输入网络，很容易造成不稳定。我们可将其初始化到 [0, 1] 或 [-1, 1] 区间，或将其处理为均值为 0、标准差为 1。



图 3-9 图像增强

进一步的预处理称为白化 (whitening)，常用方法包括 PCA 和 ZCA 白化。对于图像还可进行直方图均衡 (histogram equalization) 等。不过，由于后文的 BN 层可在一定程度上自动完成类似的工作，因此在使用 BN 层后，不一定需要再对图像做精细的预处理。

3.3 神经网络的调参

网络的学习速率、批大小等称为网络的超参数 (hyper-parameters)。在广义上，可将网络参数的初始化方法、梯度下降算法的选择、网络的层数、每层的神经元数等也称为超参数。

超参数的设置不但会影响训练的速度，也会影响训练的准确度，因此我们在训练时会经常调整超参数，这简称为调参。

调参其实并不困难。如果计算资源充足，可通过自动的方式完成，如使用网格搜索 (grid search) 或随机搜索，甚至可用另一个机器学习模型 (例如，另一个神经网络) 预测最佳的超参数。

3.3.1 学习速率

回顾此前的梯度下降公式：

$$w^{\text{new}} = w - \eta \cdot \frac{\partial \text{LOSS}}{\partial w}$$

其中 η 是学习速率，它类似于梯度下降时的步长，需通过实验决定。通常可先实验 0.1，如果发现太大就实验 0.01，依此类推。

可用简单的损失 LOSS 的等高线图，如图 3-10 所示，说明学习速率对于学习过程的影响：

- 图 3-10a 是学习速率合适的例子。
- 图 3-10b 是学习速率太大的例子，由于每步迈得太大，出现了明显的振荡，而且可能让网络崩溃，导致损失变为 NaN（浮点数错误）。

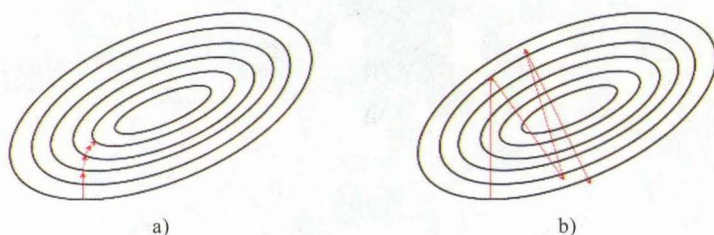


图 3-10 学习速率的影响：直观解释

学习速率如果太小，也不利于训练，训练的速度会更慢，也更可能陷入局部最小值，出现欠拟合。我们可将这些情况总结为如图 3-11 所示。

由图 3-11 可见，最佳方法是在训练的初始阶段使用较高的学习速率（绿线），然后缩小学习速率（红线），这样一开始的训练速度快，而且最后的收敛效果好。

逐渐减少学习速率，相当于逐渐减少梯度下降的每步步长，可改善最终的收敛效果，如图 3-12 所示。

在实际训练中，通常采取这样的方案：最初 X 个 epoch 使用 a 的学习速率，然后每过 Y 个 epoch 将学习速率减少为原来的 b 分之一。

学习速率也与批大小有关。如果将批大小设为原来的 N 倍，那么适合将学习速率设为原来的 N^α 倍，其中 α 的取值通常在 0.5 到 1 之间，具体需由实验决定。下文在讨论批大小时会进一步讨论。

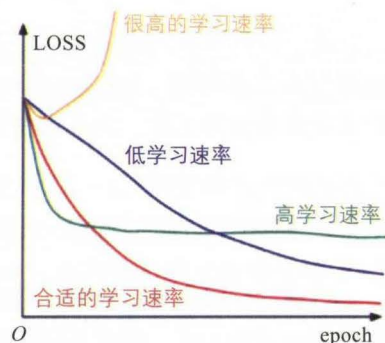


图 3-11 学习速率的影响：训练精度曲线

3.3.2 批大小

如前所述，如果每次随机选 N 个样本，然后对它们对应的梯度取平均，用于这次的权重更新，那么批大小就是 N 。

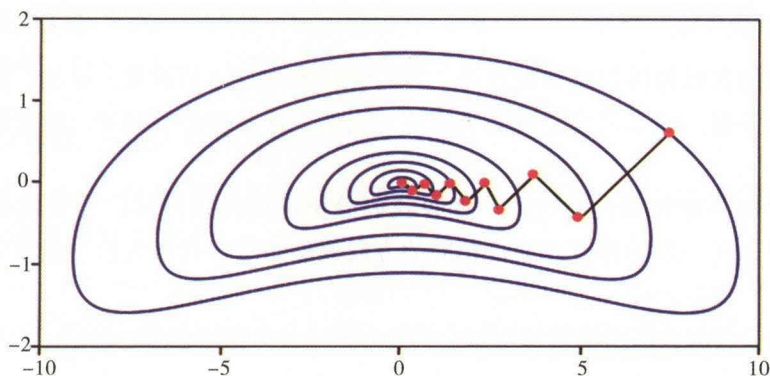


图 3-12 逐渐减少学习速率

如果批大小取更大的值：

- 优点：训练速度快。如前所述，前向传播和反向传播都可写成矩阵形式，矩阵越大，完成全体训练集的总时间就会越少。
- 缺点：占用显存大，因此需要更昂贵的显卡。
- 缺点：此时 SGD 中的梯度会更准确，因为使用了更多样本取平均。但网络更容易陷入鞍点和极值点的泥潭，最终得到的网络性能往往不如批大小较小时的效果好。

如果批大小取更小的值：

- 缺点：训练速度慢。
- 优点：占用显存小。
- 优点：此时梯度的随机性（噪音）更大，但网络更容易跳出鞍点和极值点，不断进步，最终性能往往更好。

研究人员的经验是，对于常见问题，最优的批大小往往在 16 到 256 之间。太小则训练过慢，太大则性能不佳。

而 Facebook 在 2017 年的论文《Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour》[⊖]中提到，通过设置更合理的学习速率，在批大小很大时，也能取得较好的性能，从而可使用高达 8192 的批大小，显著缩短训练时间。

具体方法是，如果将批大小提高到原来的 N 倍，那么可将学习速率 η 也设为原来的 N 倍，但并不是马上设为 N 倍，而是加入一个热身过程：初始的 η 仍然是原来的大小，在训练过程中不断提高 η ，直到 X 个 epoch 后才达到原来的 N 倍。最后，随着训练的进行，再逐渐缩小 η ，与此前的方式一致。

Facebook 的后续研究《Large Batch Training of Convolutional Networks》[⊖]中提到，通过逐层调整学习速率，还可进一步提高批大小。

⊖ 地址为 <https://arxiv.org/abs/1706.02677>。

⊖ 地址为 <https://arxiv.org/abs/1708.03888>。

3.3.3 初始化方法

神经网络的参数包括权重和偏置等，它们的初始化方法很重要。举例，如果初始参数就接近最终的参数，那么就可立即完成训练。当然，这是理想的例子，但我们可试图接近这个目标。

首先，参数不能初始化为全零。如果我们分析反向传播过程，会发现如果参数初始化为全零，那么同一层的神经元会往相同的方向调整参数，这就失去了使用多个神经元的意义。

因此，通常我们会将网络的初始权重设置为正态分布的随机数（网络的初始偏置仍然可设置为全零）。

如图 3-13 所示，如果神经元采用随机初始化，相当于从不同位置出发，会有不同的演变方向，于是最终会分别识别不同的目标，提高网络的性能。

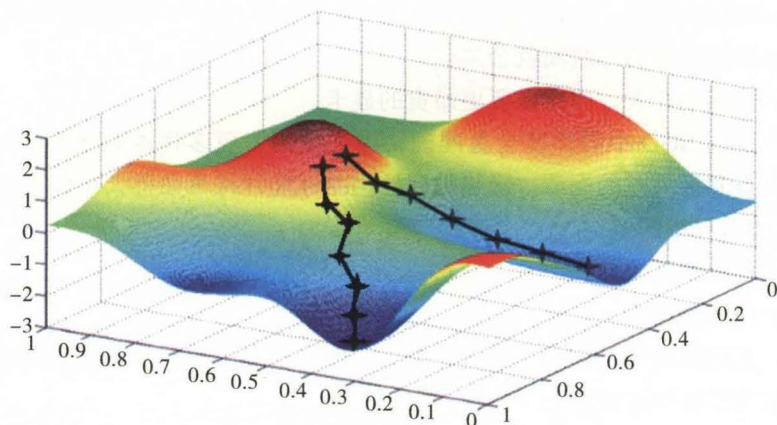


图 3-13 梯度下降

这就像在丘陵地带下山，如果一开始大家都在一起，那么最终会下到同一个山谷；但如果一开始大家是随机分布的，那么就会最终到达各个不一样的山谷。

这里的另一个问题是，随机初始化所使用的正态分布的标准差如何设置？如果进一步分析反向传播过程，我们会发现，每个神经元的初始化方法应与这个神经元的输入和输出连接个数有关。具体做法可参见 MSRA 的论文《Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification》^①。这种初始化方法在 MXNet 中称为 MSRAPrelu，可直接调用。在《All you need is a good init》^②中提供了一种更精细的初始化方法，而在《The Shattered Gradients Problem》^③中还有一种使用了残差网络（后文会介绍）思想的有趣初始化方法。

① 地址为 <https://arxiv.org/abs/1502.01852>。

② 地址为 <https://arxiv.org/abs/1511.06422>。

③ 地址为 <https://arxiv.org/abs/1702.08591>。

最后，是否有办法可尽量接近“在初始化就一步到位”的梦想？这可通过预训练（pre-training）与精调（fine tuning）实现，我们会在后文叙述。

3.3.4 调参实战：重返 TensorFlow 游乐场

让我们回到 TensorFlow 游乐场^①，如图 3-14 所示，其中左边第四个分类问题（螺旋型）最复杂，也是在此需解决的问题。

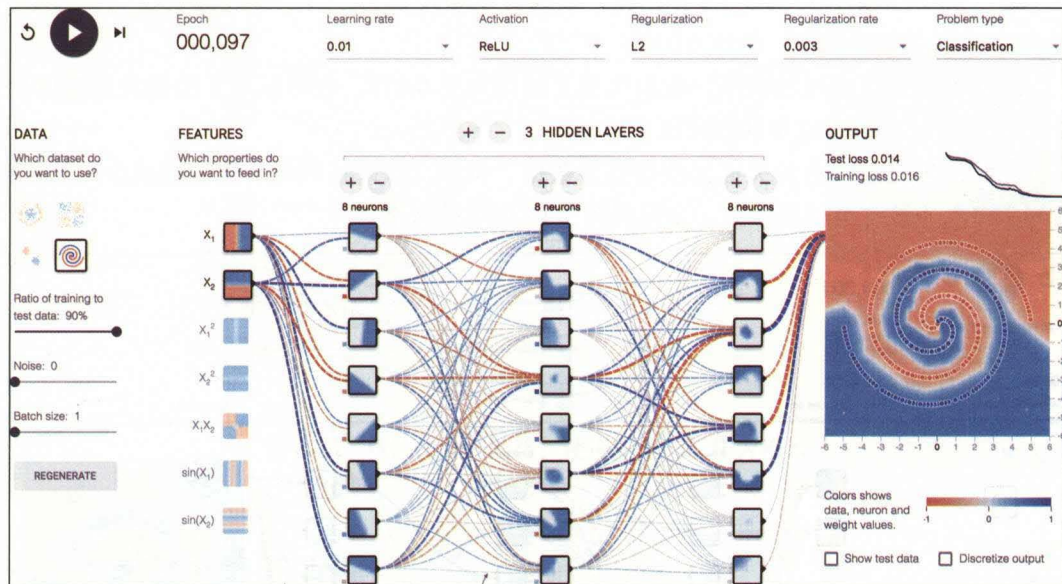


图 3-14 在 TensorFlow 游乐场运行较复杂的网络

上图中的蓝色代表正，白色代表 0，黄色代表负。

对于每个神经元的图像， X 轴代表 x_1 ， Y 轴代表 x_2 。其中某个点的颜色，代表在 x_1 和 x_2 取这个点所对应的值时，这个神经元的输出为非线性函数后的值。举例：

- ☐ 这里图中的非线性是 ReLU，所以所有神经元的图像都只包括白色和蓝色，因为 ReLU 的输出是非负的。
- ☐ 代表 x_1 的图像是左黄右蓝，因为 X 轴是左负右正。
- ☐ 代表 x_2 的图像是下黄上蓝，因为 Y 轴是下负上正。

水管的颜色是权重，左下角的小方块是偏置。例如，对于第 1 层神经元的第 2 个神经元：

- ☐ 它和 x_1 之间的权重是黄色，负值，所以它的图像大致是左蓝右白。
- ☐ 它和 x_2 之间的权重是蓝色，正值，所以它的图像大致是上蓝下白。

① 地址为 <http://playground.tensorflow.org/>。

□ 它的偏置是负值，所以会稍微减少图像中蓝色的面积。

这些因素综合作用，最后输出的图像是左上蓝，其他部分白。

如果只使用 x_1 和 x_2 这两个输入，如何训练出性能优秀的网络？要点如下：

- 较大的网络规模：例如 3 层网络，每层 8 个神经元。
- 较多的数据：例如使用数据的 90% 作为训练集，10% 作为测试集。
- 较小的批大小：例如 1。
- 一定的正则化：例如 L2 正则化，系数为 0.003。
- 合适的学习速率：例如 0.01。

以此可训练出前图的网络，在训练集上的损失是 0.016，在测试集上的损失甚至更少，只有 0.014（这是因为这里的问题较为简单）。

在此，避免过拟合的关键是使用更多数据。例如，如果将训练所用数据减少到总数据集的 30%，不改变其他超参数，会得到图 3-15 所示的结果。

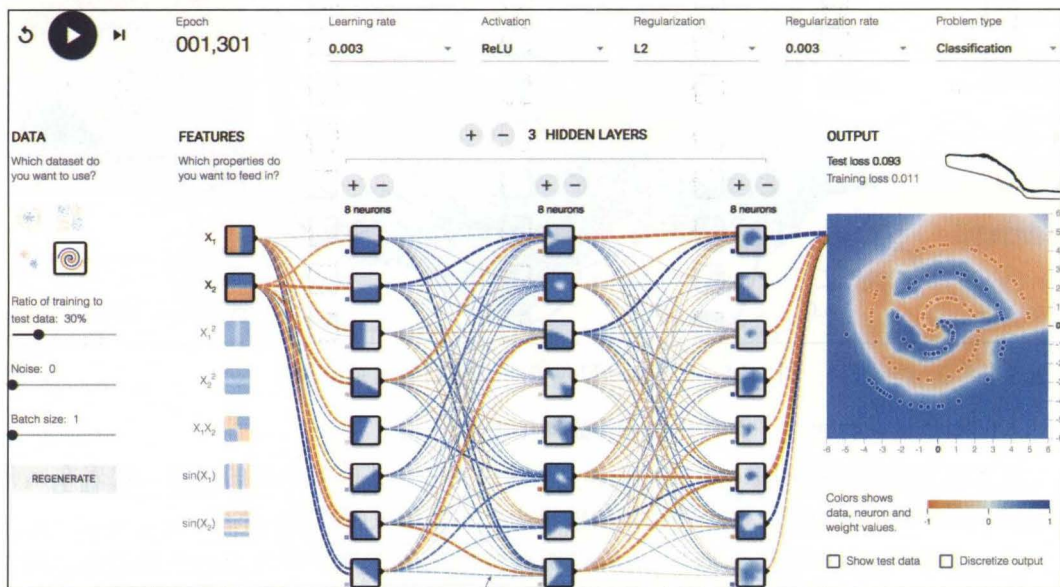


图 3-15 减少训练数据，会造成过拟合

最终的训练损失为 0.011，测试损失高达 0.093，这说明出现了明显的过拟合。

最后，我们用实例说明深度神经网络的特点。在深度学习发展之前，研究人员往往会使用多个特征作为模型的输入，并使用浅层模型。例如，如图 3-16 所示，使用 7 种特征输入，并只使用 1 个隐层。

与此前的网络比较，我们会看到此前更深的网络的中间层可自动发现和构造出许多特征，因此无须人工选择大量特征作为网络输入。

在 AlphaGo 的策略网络中也可以看到类似的设计。在旧版 AlphaGo 中，策略网络使用

多个人工构造的特征层作为输入。而在新版 AlphaGo 中，只使用简单的特征层作为输入，由更深的网络自动发现特征。



图 3-16 传统的神经网络：使用多个特征作为输入

3.4 实例：MNIST 问题

本节的目的是用全连接网络训练 MNIST 手写数字识别网络。MNIST 是机器学习中的经典数据集，其主页和下载地址为：<http://yann.lecun.com/exdb/mnist/>。

MNIST 数据集中有 70000 个手写数字（从 0 到 9）图像。其中 60000 个属于训练集，10000 个属于测试集。这些数字以 28×28 的灰度图像存储，每个图像中只有 1 个数字，样例如图 3-17 所示。



图 3-17 MNIST 中的数字实例

网络的目标是识别出输入的手写数字图像中的数字。

需要指出，MNIST 有一个缺点：它比较简单，很容易达到较高的准确率，因为每个数字所对应的图像都较为接近，例如用一个简单的线性分类器就可达到 88% 的识别准确率。

虽然如此，MNIST 中有少量数字很潦草，模棱两可，在人类看来也很难辨别，因此实现 100% 的识别准确率几乎不可能。目前最佳神经网络模型的识别准确率在 99.8% 左右，这与人类的识别准确率已经相当。

在此我们先用全连接神经网络实验，可达到 98% 的准确率。后文我们将用卷积网络实现准确率 99.4% 的模型。

3.4.1 重要知识：SoftMax 层、交叉熵损失

在图像分类时，网络的目标是输出图像属于每个分类的概率，例如，99% 的可能性是猫，0.6% 的可能性是狗，0.1% 的可能性是马……

这有利于网络的训练和评估。因为如果正确答案是马，那么“40% 是马，60% 是狗”就比“20% 是马，80% 是狗”更好。

而且，有时图像中会有多个对象分类并不明确，我们也确实应输出它属于每一个分类的概率。例如，对于图 3-18 所示的图像，很难说它应分类到猫还是狗。



图 3-18 难以分类的图像

这里的分类概率需满足 2 个条件：

- 每一个分类的概率都应该在 0% 和 100% 之间。
- 由于每张图像只能属于一个分类，因此所有分类的概率之和应等于 100%。

为满足以上两点，可在最后一层使用 SoftMax 作为非线性激活函数。若上一层的输出

为 $\{z_i\}$ ，则经过 SoftMax 后的输出为 $\left\{q_i = \frac{e^{z_i}}{\sum_i e^{z_i}}\right\}$ ，相当于对概率做归一化处理：

- 原始输出 $= \{z_1, z_2, \dots, z_n\}$ ，每个数字不一定在 0 到 1 之间，相加不一定为 1。
- SoftMax 输出 $= \left\{\frac{e^{z_1}}{e^{z_1} + e^{z_2} + \dots + e^{z_n}}, \frac{e^{z_2}}{e^{z_1} + e^{z_2} + \dots + e^{z_n}}, \dots, \frac{e^{z_n}}{e^{z_1} + e^{z_2} + \dots + e^{z_n}}\right\}$ ，每个数字一定在 0 到 1 之间，相加一定为 1。

举例：

- 原始输出 $= \{2, -3, 4\}$

□ SoftMax 输出 = $\left\{ \frac{e^2}{e^2 + e^{-3} + e^4} \approx 0.119, \frac{e^{-3}}{e^2 + e^{-3} + e^4} \approx 0.001, \frac{e^4}{e^2 + e^{-3} + e^4} \approx 0.880 \right\}$,
满足上述 2 个条件。

另外，如果每张图像可属于多个分类，即标签（tagging）问题，则总概率可大于 1，那么只需之前的第 1 个条件，不需要第 2 个条件。此时可用 sigmoid 作为输出的非线性激活函数，因为 sigmoid 的值一定是在 0 到 1 之间。

举例，如果希望输出的是：“图中有猫的概率”，“图中有狗的概率”，“图中有轮胎的概率”，那么对于前图，这 3 个概率都应该是 1。这种情况就适合用 3 个 sigmoid 神经元，输出 3 个概率。

再看损失函数。对于分类问题，损失函数推荐使用交叉熵（cross-entropy, CE）损失，又称 categorical cross-entropy loss、negative log-likelihood loss 或 log-loss。其定义是，若期望概率分布为 $\{p_i\}$ ，网络的输出为 $\{q_i\}$ ，则网络的交叉熵损失为：

$$\text{LOSS} = - \sum_i p_i \cdot \log q_i$$

读者可验证交叉熵损失为非负，且仅当 $\{q_i\}$ 与 $\{p_i\}$ 相等时为 0。交叉熵损失的一大优点是可改善梯度消失。我们会在本节最后证明。

最后，若期望输出为“ p_i 仅在 $i=n$ 时为 1”，即“属于第 n 类”，则交叉熵损失可简写为：

$$\text{LOSS} = - \log q_n$$

下面举例说明交叉熵损失的计算：

- 网络输出 = $\{0.119, 0.001, 0.880\}$
- 期望概率分布 = $\{0, 0, 1\}$
- 损失 = $-0-0-1 \cdot \log 0.880 = -\log 0.880 \approx 0.128$

上文是多个分类，多个概率的情况。有时我们只希望网络输出 1 个概率，例如在 AlphaGo 价值网络中，我们希望输出 1 个胜率。这时可用 sigmoid 作为输出的非线性激活函数，但需要修改交叉熵损失公式，使用 binary cross-entropy loss 公式，即若期望的概率为 p ，网络输出的概率为 q ，则损失为：

$$\text{LOSS} = -p \cdot \log q - (1-p) \cdot \log(1-q)$$

这相当于将网络输出看作是二分类的概率：属于 X 的概率为 q ，属于非 X 的概率为 $1-q$ 。然后将期望输出也看作是二分类的概率：属于 X 的概率为 p ，属于非 X 的概率为 $1-p$ 。再套用之前的交叉熵公式，即可得到 binary cross-entropy loss 公式。

需要指出，虽然交叉熵损失十分常用，但即使是对于涉及概率的问题，也并非一定要使用交叉熵损失，它其实与统计学中的 Kullback-Leibler 散度（简称 K-L 散度）有关。例如，在 GAN 中，有时就会使用不同的散度及对应的不同的损失。

最后，我们计算交叉熵损失的梯度情况。考虑刚才的二分类的例子，令 sigmoid 层的输入为 z ，输出为 $q = \sigma(z)$ ，则：

$$\frac{\partial \text{LOSS}}{\partial q} = \frac{\partial}{\partial q} (-p \cdot \log q - (1-p) \cdot \log(1-q)) = \frac{-p}{q} + \frac{1-p}{1-q}$$

此前算过：

$$\frac{\partial q}{\partial z} = \sigma'(z) = \sigma(z)(1 - \sigma(z)) = q \cdot (1 - q)$$

因此：

$$\frac{\partial \text{LOSS}}{\partial z} = \frac{\partial \text{LOSS}}{\partial q} \cdot \frac{\partial q}{\partial z} = q - p$$

可见，只要网络的输出 q 与期望输出 p 有区别，就一定会有一定大小的梯度。

另一方面，如果不采用交叉熵损失，例如用 MSE 损失，就容易遇到梯度消失问题。下面计算说明。MSE 损失是：

$$\text{LOSS}_{\text{MSE}} = (q - p)^2$$

则：

$$\frac{\partial \text{LOSS}_{\text{MSE}}}{\partial z} = \frac{\partial \text{LOSS}_{\text{MSE}}}{\partial q} \cdot \frac{\partial q}{\partial z} = 2(q - p) \cdot q \cdot (1 - q)$$

那么，若 q 非常接近 0，或 q 非常接近 1，梯度就会非常接近 0，这就是梯度消失。

同样，对于 SoftMax 与交叉熵损失的情况，也可计算出：

$$\frac{\partial \text{LOSS}}{\partial z_i} = q_i - p_i$$

由此可见此时也成功避免了梯度消失。这个计算略为烦琐，但建议有能力的读者验算。计算过程会使用概率的归一化条件：

$$\sum_i p_i = 1$$

3.4.2 训练代码与网络架构

在此我们将开始用 GPU 加速训练。读者可下载 CPU-Z 和 GPU-Z 软件，观察训练时 CPU 和 GPU 是否在满负荷工作，是否存在瓶颈。注意内存和 IO 的速度也可能成为瓶颈。

首先，下载 MNIST 的数据和标签文件，并将它们放到与训练代码相同的目录：

- ❑ <http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz>
- ❑ <http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz>
- ❑ <http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz>
- ❑ <http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz>

上述文件均为 gzip 压缩格式，我们会在 Python 代码中解压。解压后的数据格式可参见

<http://yann.lecun.com/exdb/mnist/>。

下列代码推荐在 Jupyter 演算本环境运行，因为其中会显示一些图像。

载入库：

```
import numpy as np
import os
import gzip
```



```
import struct
import logging
import mxnet as mx
import matplotlib.pyplot as plt # 这是常用的绘图库
```

读入数据并设置迭代器：

- 注意 MXNet 中图像的数组格式是 [样本, 通道, 宽, 高]。对于这里的灰度图像, 通道数为 1。
- 数据的归一化很重要。建议数据的绝对值不超过 1, 有助于提高梯度的稳定性。

```
logging.getLogger().setLevel(logging.DEBUG)
```

```
def read_data(label_url, image_url): # 读入训练数据
    with gzip.open(label_url) as flbl: # 打开标签文件
        magic, num = struct.unpack(">II", flbl.read(8)) # 读入标签文件头
        label = np.fromstring(flbl.read(), dtype=np.int8) # 读入标签内容
    with gzip.open(image_url, 'rb') as fimg: # 打开图像文件
        magic, num, rows, cols = struct.unpack(">IIII", fimg.read(16)) # 读入图像
        # 文件头, rows和cols都会是28
        image = np.fromstring(fimg.read(), dtype=np.uint8) # 读入图像内容
        image = image.reshape(len(label), 1, rows, cols) # 设置为正确的数组格式
        image = image.astype(np.float32)/255.0 # 归一化为0到1区间
    return (label, image)
```

```
# 读入数据
(train_lbl, train_img) = read_data('train-labels-idx1-ubyte.gz', 'train-images-idx3-ubyte.gz')
(val_lbl, val_img) = read_data('t10k-labels-idx1-ubyte.gz', 't10k-images-idx3-ubyte.gz')
```

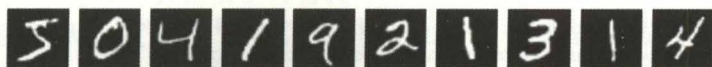
```
batch_size = 32 # 批大小
```

```
# 迭代器
train_iter = mx.io.NDArrayIter(train_img, train_lbl, batch_size, shuffle=True)
val_iter = mx.io.NDArrayIter(val_img, val_lbl, batch_size)
```

显示其中的数据样例：

```
for i in range(10): # 输出前10个数字
    plt.subplot(1,10,i+1) # 这里的语句可参见matplotlib库的介绍
    plt.imshow(train_img[i].reshape(28,28), cmap='Greys_r')
    plt.axis('off')
plt.show() # 显示图像
print('label: %s' % (train_lbl[0:10],)) # 显示对应的标签
```

输出效果如图 3-19 所示。



```
label: [5 0 4 1 9 2 1 3 1 4]
```

图 3-19 数据样例

定义网络：

```
data = mx.symbol.Variable('data')

# 将图像摊平，例如1*28*28的图像会变为784个数据点，这样才可与普通神经元连接
flatten = mx.sym.Flatten(data=data, name="flatten")

# 第1层网络及非线性激活，有128个神经元，使用ReLU非线性
fc1 = mx.sym.FullyConnected(data=flatten, num_hidden=128, name="fc1")
act1 = mx.sym.Activation(data=fc1, act_type="relu", name="act1")

# 第2层网络及非线性激活，有64个神经元，使用ReLU非线性
fc2 = mx.sym.FullyConnected(data=act1, num_hidden=64, name="fc2")
act2 = mx.sym.Activation(data=fc2, act_type="relu", name="act2")

# 输出神经元，因为需分为10类，所以有10个神经元
fc3 = mx.sym.FullyConnected(data=act2, num_hidden=10, name="fc3")
# SoftMax层，将上一层输出的10个数变为10个分类的概率
net = mx.sym.SoftmaxOutput(data=fc3, name='softmax')
```

显示网络的参数情况：

```
# 我们将调用MXNet中的viz库，需要先告知MXNet输入数据的格式
shape = {"data" : (batch_size, 1, 28, 28)}
mx.viz.print_summary(symbol=net, shape=shape)
```

输出为：

Layer (type)	Output Shape	Param #	Previous Layer
data(null)	1x28x28	0	
flatten(Flatten)	784	0	data
fc1(FullyConnected)	128	100480	flatten
act1(Activation)	128	0	fc1
fc2(FullyConnected)	64	8256	act1
act2(Activation)	64	0	fc2
fc3(FullyConnected)	10	650	act2
softmax(SoftmaxOutput)	10	0	fc3
Total params: 109386			

读者可验算这里的参数量。例如，fc1 共有 128 个神经元，每个神经元的输入是 784 个数据点，所以每个神经元有 784 个权重，还有 1 个偏置，所以 fc1 层的参数量为 $128 \times (784 + 1) = 100480$ 。

显示网络的结构图：

```
mx.viz.plot_network(symbol=net, shape=shape).view()
```

输出效果如图 3-20 所示。

训练网络：

```
# 由于训练数据量较大，这里采用了GPU，若读者没有GPU，可修改为CPU
module = mx.mod.Module(symbol=net, context=mx.gpu(0))

module.fit(
    train_iter,
    eval_data=val_iter,
    optimizer = 'sgd',
```

采用0.2的初始学习速率，并在每60000个样本后（即每1个epoch后）将学习速率缩减为之前的0.9倍

```
optimizer_params = {'learning_rate': 0.2, 'lr_scheduler'
                    : mx.lr_scheduler.FactorScheduler(step=60000/batch_
                    size, factor=0.9)},
num_epoch = 20,
batch_end_callback = mx.callback.Speedometer(batch_
size, 60000/batch_size)
```

训练过程如下：

```
INFO:root:Epoch[0] Train-accuracy=0.752017
INFO:root:Epoch[0] Time cost=3.199
INFO:root:Epoch[0] Validation-accuracy=0.939996
INFO:root:Update[1876]: Change learning rate to 1.80000e-01
INFO:root:Epoch[1] Train-accuracy=0.958383
INFO:root:Epoch[1] Time cost=3.194
INFO:root:Epoch[1] Validation-accuracy=0.953175
.....
INFO:root:Update[35626]: Change learning rate to 2.70170e-02
INFO:root:Epoch[19] Train-accuracy=0.999933
INFO:root:Epoch[19] Time cost=3.204
INFO:root:Epoch[19] Validation-accuracy=0.980531
```

由于使用了较好的批大小和学习速率，最终的测试集准确率达到 98%。通常的全连接网络样例只能达到 96% 左右。

由于在训练集的性能高达 99.99%，因此这里的主要障碍是过拟合。

后续我们将使用卷积网络再次实验，将可实现更少的参数量，且在测试集的性能会更好。

3.4.3 超越 MNIST：最新的 Fashion-MNIST 数据集

许多研究人员认为，MNIST 数据集过于简单，而且只是抽象的数字形状，与现实图像的区别很大。GAN 的发明者 Ian Goodfellow 已号召大家尽量少用 MNIST，而 Keras 的创始人 Francois Chollet 也深为赞同，表示 MNIST 的代表性实在不佳。

然而 MNIST 数据集是如此深入人心，积重难返，即使在 2017 年，它仍然是机器学习文献中最常使用的数据集，而且使用量还在不断上升。

在 2017 年 8 月，研究人员提出了可替代 MNIST 的数据集——Fashion-MNIST^①，它比 MNIST 的难度更大，更有代表性，且与 MNIST 的数据结构完全相同，同样是 60000+10000 张 28×28 的灰度图像，目标也是分为 10 类，因此可直接替换原有的 MNIST 数据，无须修改任何代码，就可测试模型在这个更好的数据集上的性能。

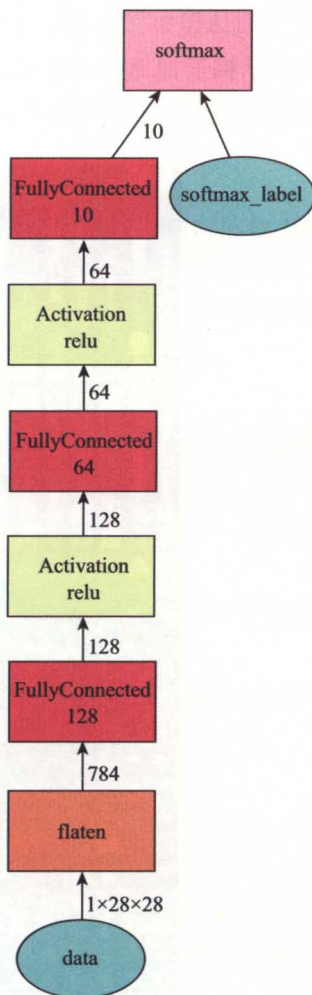


图 3-20 网络结构图

① <https://github.com/zalandoresearch/fashion-mnist>

具体说来, Fashion-MNIST 的目标是将 28×28 的灰度服饰图像分成 10 类: 0 表示 T 恤, 1 表示裤子, 2 表示套头衫, 3 表示连衣裙, 4 表示外套, 5 表示凉鞋, 6 表示衬衫, 7 表示运动鞋, 8 表示袋包, 9 表示靴子, 如图 3-21 所示。其中有些类别很相似, 人类也需要仔细观察。

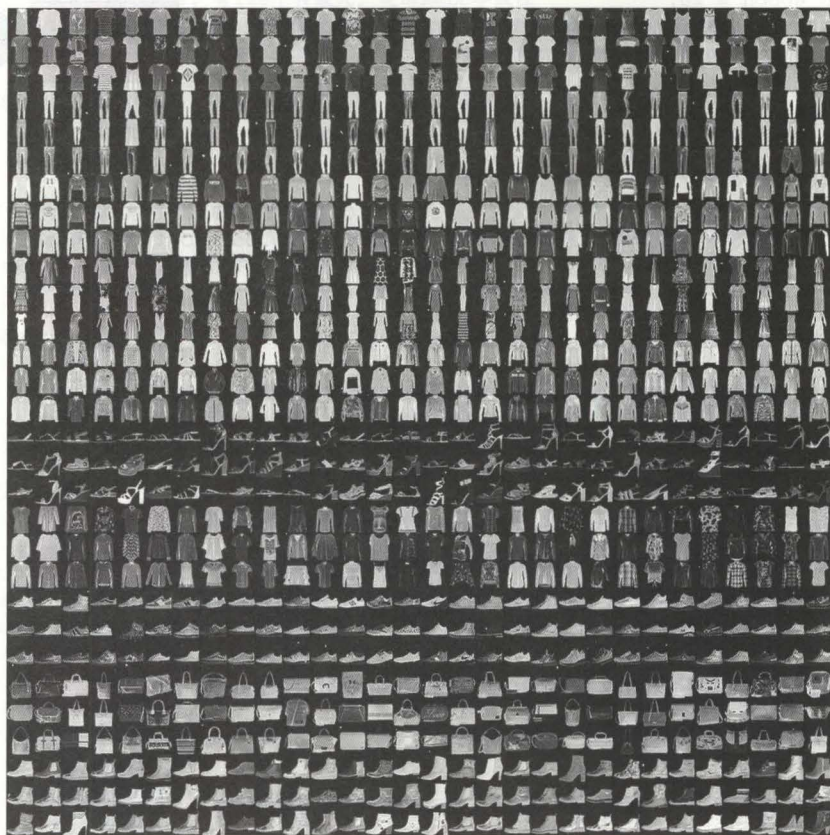


图 3-21 Fashion-MNIST 中的图片

下面看效果。下载 Fashion-MNIST 数据, 替换之前的 MNIST 数据, 然后运行之前的代码, 训练情况如下:

```
INFO:root:Epoch[0] Train-accuracy=0.681483
INFO:root:Epoch[0] Time cost=3.176
INFO:root:Epoch[0] Validation-accuracy=0.836262
INFO:root:Update[1876]: Change learning rate to 1.80000e-01
INFO:root:Epoch[1] Train-accuracy=0.842167
INFO:root:Epoch[1] Time cost=3.202
INFO:root:Epoch[1] Validation-accuracy=0.853534
.....
INFO:root:Update[35626]: Change learning rate to 2.70170e-02
INFO:root:Epoch[19] Train-accuracy=0.931450
INFO:root:Epoch[19] Time cost=3.165
INFO:root:Epoch[19] Validation-accuracy=0.892772
```

确实难度更大，最后的训练准确率为 93% 左右，验证准确率为 89% 左右。我们在后文会改进这个性能。

3.5 网络训练的常见 bug 和检查方法

在本章的最后，我们介绍关于网络训练更多的实际经验。如果读者遇到不明白的地方，可先跳过这些地方，在阅读完后文，更熟悉网络的训练过程后，再回头来看。

如果网络在训练集的性能一直很低，例如准确率只有 50% 或更低，我们就应考虑程序中是否存在 bug。

第一种 bug 来自数据，检查方法如下：

- ❑ 对数据做可视化（visualization），检查数据是否正确，尤其是格式是否正确、宽度和高度是否翻转了、训练数据和测试数据的格式是否一致。
- ❑ 检查每个 batch 是否正确地使用了随机次序的数据。
- ❑ 检查数据迭代器是否正常运作。
- ❑ 检查数据的增强代码和预处理代码是否正确。注意训练和测试时应使用相同的预处理代码，例如应归一化到同一区间。
- ❑ 检查标签是否正确，在打散数据顺序时是否相应打散了标签。需要保证标签和数据正确对应。

第二种 bug 来自模型，检查方法如下：

- ❑ 检查在 MXNet 定义网络时，是否写错了变量名、神经元个数、卷积核大小等，或是否写漏了层，例如漏了 ReLU 层。
 - 如果只是简单地复制粘贴各个层，然后手动修改每层的定义，很容易出现这种问题。因此，推荐用函数的方法定义层（后文在 MXNet 的使用技巧章节会介绍），这样出错的概率低，代码容易改动，更简洁清晰。
 - 可通过 `mx.viz.plot_network` 输出网络的流程图，快速检查是否存在错误。
- ❑ 检查模型是否能完全拟合一个很小的训练数据集。例如，仅使用 100 个样本训练，网络应能达到接近 100% 的准确率。
- ❑ 检查模型是否能完全拟合一个非常简单的目标。例如，如果手工设置所有图像都属于同一个分类，网络应能立刻达到 100% 准确率。
- ❑ 检查损失函数的计算是否正确。人工检查可视化网络的输出与正确标签。还可加入更多的性能指标（在 MXNet 中为 `metric`）。

第三种 bug 来自训练算法，常见的错误包括：算错导数、算错符号、算错公式。如果手工写训练程序、手工定义特殊的层、手工定义特殊的运算（例如新的 ReLU 变种），就可能出现这种错误。

此时，读者可先检查最简单的数据和网络结构，以确定训练算法是否有 bug。如前所

述，神经网络的拟合能力很强，它应能完美拟合简单的数据和目标。如果网络做不到这一点，就说明很可能训练算法存在错误。

3.6 网络训练性能的提高

说到提高性能，大家可能会说调参，但这里其实还有更多的技巧。

可观察网络的输出：

- ❑ 对网络的输出做可视化，观察在何种情况下正确，在何种情况下错误，正确和错误的程度如何。
 - 例如，如果某张图像中同时包括猫和狗，那么网络将这张图像识别为猫或狗都可接受，并不是真正的错误。
 - 但是如果网络将一张明显看上去是猫的图像识别为狗，那就值得寻找原因了。
- ❑ 值得特别观察的是，在何种情况下网络的错误最严重，这往往可提供重要的启示。
 - 这可发现某些隐蔽的问题，包括训练数据集的错误、如数据集本身的标注错误、剪裁不当等。

可尝试改善数据：

- ❑ 数据增强和预处理都很重要。需注意。
- ❑ 在测试时通常会关闭数据增强。如还需数据增强，可将多张处理后的图像送入网络，然后取输出的均值作为输出。
- ❑ 使用过多的数据增强会导致欠拟合。使用过少的数据增强可能会导致过拟合。建议对增强后的数据做可视化，观察数据增强是否过度或不够。
- ❑ 在训练时应打散数据的次序。
- ❑ 数据的数量是否不够？本身的质量是否不佳？
 - 数据越多越好。如果确实缺乏数据，可尝试预训练，或转用其他机器学习方法（如 SVM 等）。
 - 数据应有一定的均衡性和代表性。例如，对于分类问题，每个分类都应有数量较为接近的训练图像。
- ❑ 如果数据的噪声太大、难度太高，网络不一定能学会数据。例如，如果试图用神经网络直接预测明天的股票涨跌，往往难以得到显著的结果。

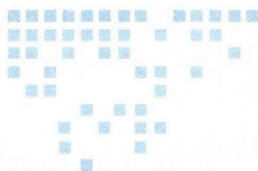
可尝试改善网络模型的设计：

- ❑ 使用更大、更深、更宽、更多路径、更先进的网络架构，往往可带来性能的提升。如后文将会介绍的 ResNet、Xception 等架构；BN、 3×3 卷积、 1×1 卷积等技巧。
- ❑ 注意正则化需适度。例如，L2 正则化和 Dropout 正则化的幅度不可太大，否则容易造成欠拟合。建议先保证网络在训练集的性能足够好（解决欠拟合），再试着提升网络在测试集的性能（解决过拟合）。

- ❑ 选取合适的损失函数，特别是在使用多任务学习或有多个预测目标的时候。
- ❑ 选取合适的性能监测指标。

可尝试改善训练方法：

- ❑ 调参。选取合适的学习速率、批大小、初始化方法等。
 - 试着把学习速率乘以 10，以及乘以 0.1，观察性能的变化。
 - 可尝试多种学习速率的衰减方法。
 - 如果网络的输出出现 NaN，往往说明学习速率过大。
 - 较小的批大小往往可带来较好的性能。
 - 如需得到最高性能，可对超参数进行全面搜索。
- ❑ 找到合适的优化器，如实验 SGD、带动量的 SGD、Adam 等。
- ❑ 多一些耐心。
 - 复杂的网络，往往会遇到“平台期”，期间的性能增长较慢。因此，只要能观察到性能在提高，不妨多等一会。
 - 更高的学习速率可能会加快训练速度。
- ❑ 观察训练的细节，如检查梯度、激活情况、权重等。
 - 可输出网络每层这些情况的直方图，
 - 最理想的情况是它们均符合高斯分布。如果出现过大的差异，就应考虑网络架构和超参数的设置是否存在问题。
 - 检查是否存在梯度消失和梯度爆炸。
 - 检查是否存在“死 ReLU”问题。
 - 检查激活值和权重是否存在过大和过小的情况。
- ❑ 最后，如果感觉训练速度过慢，可下载 CPU-Z 和 GPU-Z 软件，观察训练时 CPU 和 GPU 是否在满负荷工作，还是存在瓶颈。注意内存和 IO 的速度也可能成为瓶颈。在程序中使用多线程往往可提高硬件的利用率。



深度卷积网络：第三课

4.1 卷积网络：从实例说明

在上节我们讨论了传统的全连接神经网络。在现代的图像处理以及 AlphaGo 中，实际上使用的是卷积神经网络。

为何不使用全连接网络？在读完上节后，读者可能会设想这样的一种围棋策略网络架构，如图 4-1 所示。

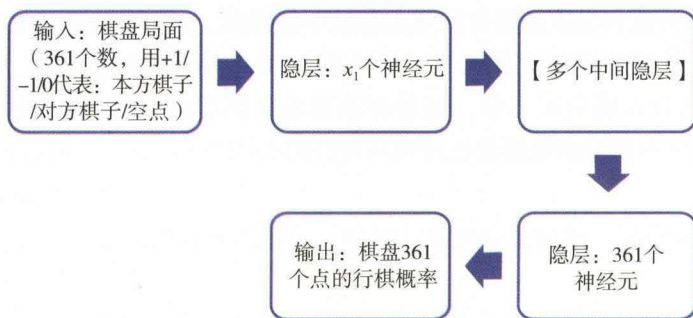


图 4-1 用全连接网络实现策略网络

但如果读者实验训练，会发现效果不佳。实际上，全连接网络并不适合处理大部分问题：

- ❑ 深层的全连接网络，很难直接训练，准确率容易出现上升停滞的现象。学术界为此曾发展过许多技术，如预训练、分层训练等。
- ❑ 即使能成功将全连接网络在测试数据中训练到较好的效果，我们也会发现它容易出现过拟合。这可以通过 Dropout 等正则化方法改善，不过最终性能仍然会不如卷积神经网络。

- ❑ 对于图像问题，全连接网络的参数量巨大，不但占用很多空间，也是它容易过拟合的原因之一。

如果使用卷积神经网络，就可以直接训练，而且网络的性能明显更好：

- ❑ 卷积神经网络更符合图像的本质：同样的特征，无论在图像的哪个位置出现，都应该被认为是类似的特征。卷积神经网络有能力做到这一点。
- ❑ 卷积神经网络拥有一定的自我正则化能力，因为我们会对图像的每一个位置都施以相同的卷积核，于是每个卷积核都会受到图像的每一个位置的训练，相当于显著增大了训练量。
- ❑ 在同样的效果下，卷积神经网络的参数量，相较于全连接网络，明显更少。

4.1.1 实例：找橘猫，最原始的方法

我们用实例说明卷积（convolution）与卷积神经网络的思想和工作原理。这里的任务是找出一张图像中有没有橘猫，如图 4-2 所示。最原始的方法，是看图中橘色所占的面积比，例如，可以说橘色占比超过 10% 就认为图中有橘猫。

具体说来，彩色图像在电脑中是按像素（就是一个点）存储，存储的是每个点的颜色。人眼有红绿蓝（RGB）三原色。所以在电脑里，每个点是3个数字，代表这个点的红绿蓝程度。

例如，对于图 4-3 的 440×440 彩色图像，其中有
 $440 \times 440 = 193\,600$ 个点，对应 $193\,600 \times 3 = 580\,800$ 个数字。



图 4-2 找橘猫

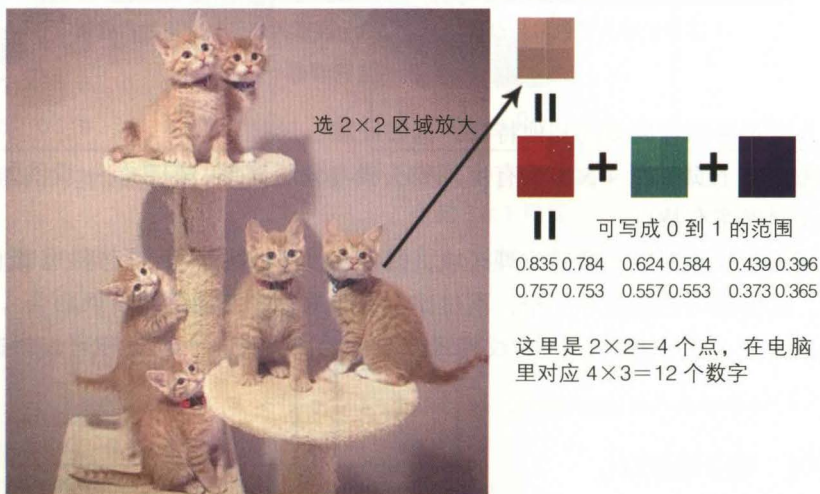


图 4-3 橘猫图片在电脑中的像素表示

如何从 580800 个数字中分辨出图像中有没有橘猫？这听上去确实有难度。最原始的方法是：计算每个点的色彩是否在橘色的色彩范围内，例如：

- 定义橘色的色彩范围是：红色在 $0.7 \sim 0.9$ ，绿色在 $0.5 \sim 0.7$ ，蓝色在 $0.3 \sim 0.5$ 。
- 然后电脑检查所有点，统计有多少个橘色点。如果橘色点的个数超过全体点的 10%，就认为有橘猫。

对图像中的每个点进行相同的操作，然后将结果汇总。这就是卷积神经网络的直观思想。我们将在下文逐步看到它的更多细节。

现在还没有出现卷积，不过如果读者熟悉卷积神经网络，就会看出，这里的方法已经可以写成一个卷积神经网络，卷积核是 1×1 ，而且网络有几层，需要用多个神经元，还需要做全局平均池化（global average pooling）。

4.1.2 实例：找橘猫，更好的方法

上节的方法，只看颜色，很容易误报和漏报，如图 4-4 所示。

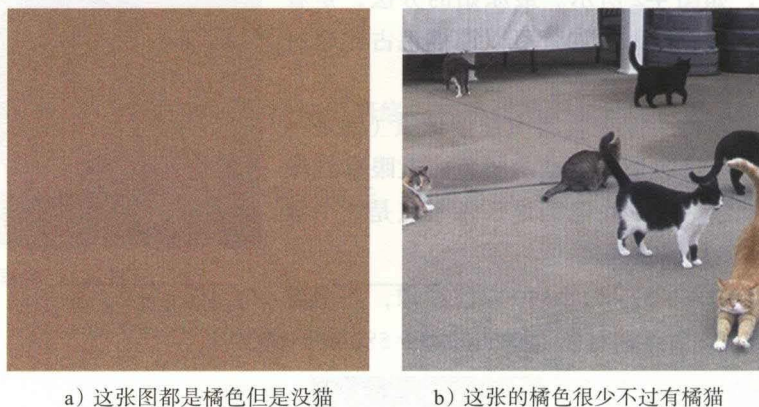


图 4-4 找橘猫不能只看颜色

更好的方法是再筛选橘猫的其他特征，例如纹理、形状。

以纹理为例，下页的图 4-5 中没有出现猫头猫身猫腿猫尾，但我们一看就知道是橘猫，因为橘猫的纹理确实有特点。

因此，可试图找到图中的橘猫纹理区域，然后计算区域的面积，达到 1% 面积就可认为图中有橘猫。这仍然可能误报和漏报，不过比上一节的方法肯定是进步了。

卷积很适合完成这个任务。因为纹理属于局部的图案特征，而卷积可以快速匹配局部的图案特征。

4.1.3 实例：卷积和池化

本节我们举例说明卷积的运作，以及它是如何实现特征匹配的。本节的目标，是找到



图 4-5 橘猫的纹理特征

像素图 4-6 中是否有“X”形图案。

我们靠肉眼可以看出，由 5 个“1”形成的 X 形位于左下角。但电脑如何能看出这一点？答案就是通过卷积的方式。

对于图中的每个点，可将它和它周围的 3×3 个点拿出来。例如，对于左上角的（第 2 行，第 2 列）点，它和它周围的 3×3 个点如图 4-7 所示。

在此定义 1 个固定的 3×3 卷积核（convolution kernel），其实就是 3×3 个数，如图 4-8 所示。

0.5	0	0	0.5	0	0.5	0	0
1	1	1	0	1	0	0	0
0	0	0	1	0.5	0.5	0	0
0	0	0	0	0.5	0.5	0	1
1	0	1	0	0	0	0.5	1
0	1	0	0	0.5	0	0	0
1	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0

图 4-6 待匹配的图片

0.5	0	0
1	1	1
0	0	0

图 4-7 （第 2 行，第 2 列）点的 3×3 局部

1	-1	1
-1	1	-1
1	-1	1

图 4-8 3×3 卷积核

将 3×3 卷积核与图 4-7 的 3×3 个点做卷积。具体方法是，先点对点相乘，然后把所有数字相加，会得到 -0.5，过程如图 4-9 所示。

$$\begin{array}{|c|c|c|} \hline 0.5 & 0 & 0 \\ \hline 1 & 1 & 1 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline 1 & -1 & 1 \\ \hline -1 & 1 & -1 \\ \hline 1 & -1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 0.5 & 0 & 0 \\ \hline -1 & 1 & -1 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \Rightarrow 0.5+0+0-1+1-1+0+0+0=-0.5$$

图 4-9 3×3 卷积操作

这里的 -0.5，就是对于这里的（第 2 行，第 2 列）点的评分。

如果再选取（第2行，第3列）的点，找出它周围的 3×3 个点，与刚才的 3×3 卷积核做卷积，会得到1.5，过程如图4-10所示。

$$\begin{array}{|c|c|c|} \hline 0 & 0 & 0.5 \\ \hline 1 & 1 & 0 \\ \hline 0 & 0 & 1 \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline 1 & -1 & 1 \\ \hline -1 & 1 & -1 \\ \hline 1 & -1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 0 & 0 & 0.5 \\ \hline -1 & 1 & 0 \\ \hline 0 & 0 & 1 \\ \hline \end{array} \Rightarrow 0+0+0.5-1+1+0+0+0+1=1.5$$

图4-10 3×3 卷积操作，另一例子

对于所有点都做这个过程（忽略图像最外圈的点，因为它周围的点不够 3×3 ），将结果按原图的顺序排列，最终效果如图4-11所示。不妨用颜色标记，看得更清楚。

-0.5	-1.5	-3.0	3.0	-1.5	1.0
1.0	-1.0	3.0	-2.0	1.0	1.0
2.0	0.0	0.0	1.0	0.5	-0.5
-3.0	2.0	0.0	-0.5	0.0	1.0
5.0	-3.0	1.5	0.5	0.0	0.5
-3.0	2.0	-0.5	-0.5	0.5	0.0

-0.5	-1.5	-3.0	3.0	-1.5	1.0
1.0	-1.0	3.0	-2.0	1.0	1.0
2.0	0.0	0.0	1.0	0.5	-0.5
-3.0	2.0	0.0	-0.5	0.0	1.0
5.0	-3.0	1.5	0.5	0.0	0.5
-3.0	2.0	-0.5	-0.5	0.5	0.0

图4-11 3×3 卷积的结果

电脑此时可直接根据每点的分数，得出结论：左下角有1个X形（评分5），上方也有类似X形的区域（评分3）。上方确实有类似X形的区域，这说明电脑对于细节的观察能力很强。

可将卷积的特点总结如下：

- 卷积可用于识别纹理和形状。不同的卷积核，可识别不同的目标。
 - 注意例子中卷积核中的-1不能改为0，否则会出错，例如会将全1也识别为X形。可见卷积核的构造是需要思考的。
 - 卷积神经网络的优点是，可通过训练自动找到合适的卷积核，无需人工考虑如何构造卷积核。
- 由例子可见，卷积操作后，图像会变小一圈。如果在 $n \times n$ 的图像上，使用一个 $k \times k$ 的卷积核，输出就会是 $(n-k+1) \times (n-k+1)$ 的图像。如果希望图像不变小，可提前给图像加一圈0（称为zero padding）。
- 在实际卷积中，往往还会加入1个偏置（bias），这个思想来自于全连接神经网络。具体方法是给得到的图像再加上1个可训练的数，也就是给每个点都加上这个数。

最后，在深度卷积网络中，会经常对卷积后的图像做进一步变换，这称为池化(pooling)，后文会继续介绍：

- 可取最大值（对于前图的例子，会得到 5.0），这标志着图中是否存在这个特征。
- 可取平均值（对于前图的例子，会得到 0.25），这标志着图中这个特征的密度。
- 可只在每个 $n \times n$ 的区域取最大值或平均值，这会得到 1 张缩小 $n \times n$ 倍的图像。

4.1.4 卷积网络的运作

在卷积网络中，会使用多个卷积核，因为不同物体、部位、角度与体积的纹理和形状不同，需用不同的卷积核去匹配。

举例说明卷积网络的运作，如图 4-12 所示。

- 将输入的 1 张图分别用 a 个卷积核处理，得到 a 张图。
- 然后将每张图加上 1 个偏置，再做非线性激活。
 - 因为卷积是一个线性操作，因此需要引入非线性激活。
 - 非线性激活的方法与在全连接网络中的一致，具体方法是在图像中的每个点调用非线性激活函数。
 - 例如，ReLU 激活的效果是：把图中所有小于 0 的数设置为 0。
 - 直观理解 ReLU 激活在此的意义：在卷积核作用后，会得到正特征和负特征，而我们往往只关心正特征。

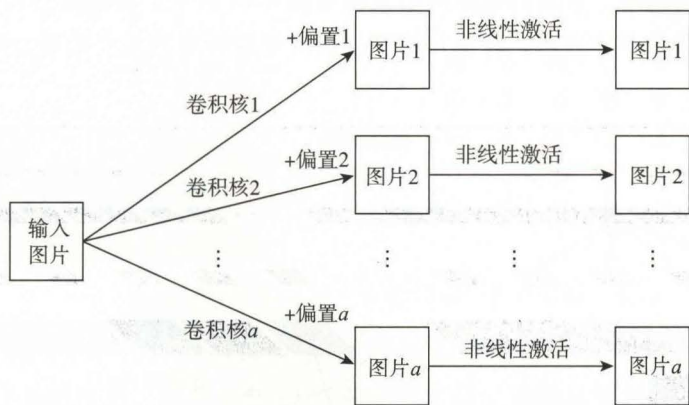


图 4-12 卷积网络的运作：输入层

得到 a 张图后，可再用 $a \times b$ 个卷积核，得到 b 张图。举例，用 $2 \times 3 = 6$ 个卷积核（不妨标记为 11 12 13 21 22 23），可将 2 张图变成 3 张图。这里的加法，是点对点相加，如图 4-13 所示。

- 然后可给每张图加上 1 个偏置，再做非线性激活。
- 然后用 $b \times c$ 个卷积核，得到 c 张图。可重复这里的过程。经过多次卷积之后，最终得到 n 张图。

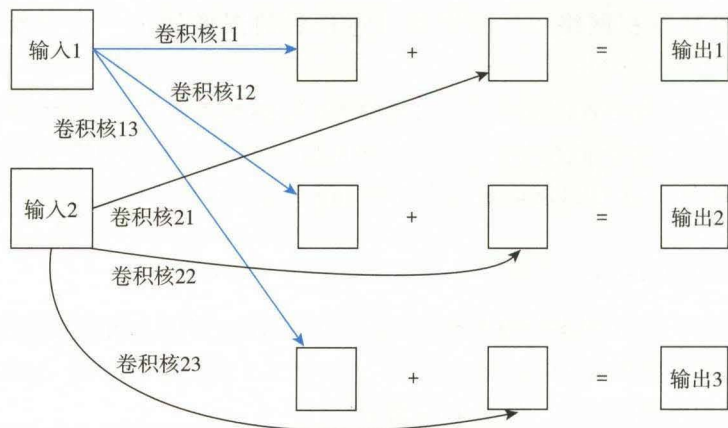


图 4-13 卷积网络的运作：中间层

卷积网络的结构，与全连接网络类似，只是把乘法换成了卷积。其实，乘法类似于 1×1 的卷积，后文会介绍。

几点补充：

- 可将 n 张图，称为 n 个通道 (channel)，或 n 个特征层 (feature plane)。
- 对于彩色图像，在电脑中会按 {R, G, B} 这 3 个通道存储，所以输入的图就已经是 3 张图。那么，在第 1 层会使用 $3 \times a$ 个卷积核，把这 3 张图变成 a 张图。

在图像分类问题中，常用的卷积网络架构如图 4-14 所示。将原图分为 {R, G, B} 通道，经卷积和 ReLU 变为多个通道，经池化层缩小，再经卷积变为多个通道，再经池化层缩小，依此类推。

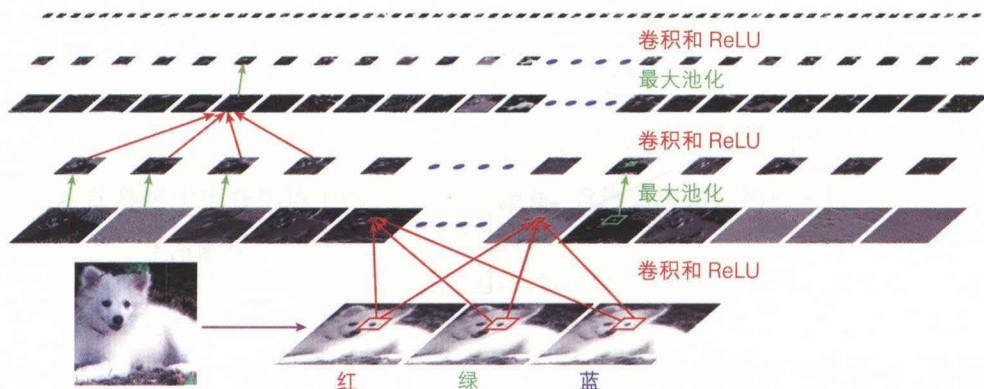


图 4-14 卷积网络的常用架构

4.2 运作：AlphaGo 眼中的棋盘

AlphaGo 的策略网络很奇妙，因为无须逻辑推理能力，只要会加法和乘法，就可运行

策略网络，达到相当强的棋力。

在此，我们看一个最简化的策略网络是如何具体运作，虽然它的棋力很低，但也能让读者体验到 AlphaGo 的感觉（策略网络的训练过程会在后文介绍，因为训练需处理棋谱，较为复杂）。

注意策略网络需使用多个特征层作为输入：

- ❑ 原版 AlphaGo 的策略网络使用 48 个特征层，包括吃子和征子的情况等。
- ❑ AlphaGo Zero 的神经网络使用 17 个特征层，其中 1 个特征层为当前方的颜色（全 1 代表黑，全 0 代表白），8 个特征层为当前方在这 1 步和前 7 步的棋子分布情况，8 个特征层为对手在这 1 步和前 7 步的棋子分布情况。这个设计更简洁，但需要配合较深的网络，否则容易出现低级失误。

本书训练的策略网络与原版 AlphaGo 类似，不过只使用其中的 8 个特征层，优点是训练速度快，且在十几层的网络上也能实现不错的棋力。

4.2.1 棋盘的编码

围棋棋盘，初看上去可编码为一张 19×19 的图像，也就是一个 19×19 数组。例如，可用 1 代表本方棋子，用 -1 代表对方棋子，用 0 代表空点。

例如对于如图 4-15 所示的这个角部（这是 AlphaGo 酷爱的“点三三”变化，这里红点代表最后的一个落子），假设本方是黑棋，可这样编码。

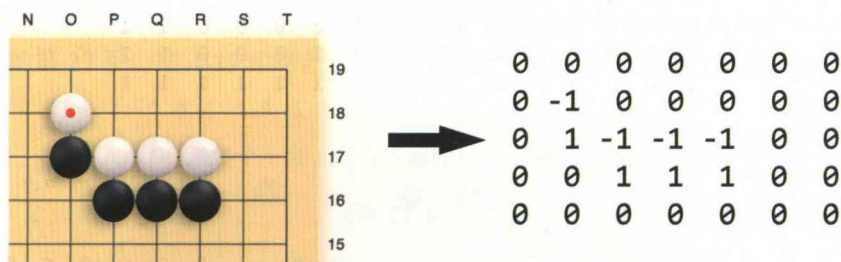


图 4-15 用 1 个 19×19 数组表示棋盘局面

但如果把这样的 1 个 19×19 数组作为网络的输入，会发现训练较慢，这有两个原因：

- ❑ 卷积操作是线性的，因此 1 和 -1 在卷积操作后，容易互相抵消，使得我们不容易单独分析某一方的情况，会受到另一方的棋子的影响。
- 注意，通过 ReLU 非线性，还是可将 1 和 -1 区分出来。
- ❑ “数气”对于围棋很重要。然而神经网络很难学会怎么精确数气（有兴趣的读者可思考这个问题）。
- AlphaGo Zero 通过使用近百层的网络，基本解决了这一问题。

在此介绍围棋的基本规则。某颗子的“气”的口数，就是它和它直接相连的同一方棋子周围的空点个数（只看上下左右方向，不包括斜线方向）。如图 4-15 所示。

□ 图中的 P17、Q17、R17 白子是直接相连的，它们周围的空点是 P18、Q18、R18、S17，所以这 3 颗白子都是有 4 口气。

□ 图中的 O18 白子是单独的 1 颗，它周围的空点是 N18、O19、P18，所以这颗白子有 3 口气。如果某方落子后，可使得对方的某些棋子没有气，就可将这些棋子“吃掉”（即，提出棋盘）。所以，时刻了解自己和对方的棋子的气的口数，非常重要。

神经网络输入的常用编码方式是“one-hot”编码，就是用 1 代表有某个性质的地方，用 0 代表没有某个性质的地方。例如，可先构造这样 3 张图，如图 4-16 所示。

- 数组一，仅存储本方棋子的情况：如果这个点是本方棋子，就设置为 1，否则设置为 0。
- 数组二，仅存储对方棋子的情况：如果这个点是对方棋子，就设置为 1，否则设置为 0。
- 数组三，仅存储空点的情况：如果这个点是空点，就设置为 1，否则设置为 0。

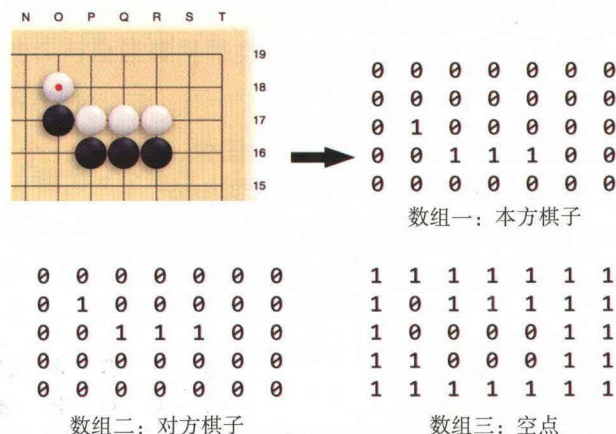


图 4-16 策略网络的输入：第 1 到 3 个数组

我们可再按气分类，加上 5 张图，如图 4-17 所示。



图 4-17 策略网络的输入：第 4 到 8 个数组

上述 8 个数组，就是我们的简化策略网络的真正输入数据。上面显示的是棋盘的局部

情况，实际输入网络的是 8 个 19×19 的数组。

4.2.2 最简化的策略网络

最简化的策略网络模型如图 4-18 所示。

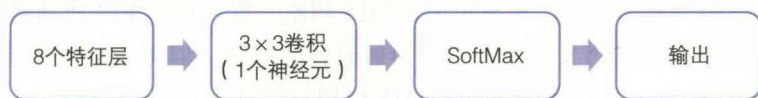


图 4-18 最简化的策略网络

整个网络只有 1 个神经元，也没有使用非线性激活函数，接近于传统机器学习中的 Logistic 回归模型。

这个网络，经过人类高手棋谱的训练，对人类高手的下法有 16.49% 的预测准确率。而且，人类高手的选择，有 25.85% 概率在模型预测的排名前 2 步内，37.44% 概率在排名前 5 步内，46.15% 概率在排名前 10 步内。

这值得肯定，因为：

- 整盘棋下来，棋盘上平均有 100 多个点可走，高手只会选择其中 1 个。所以如果瞎猜，猜对的可能性小于 1%。
- 现在我们只要做一些加法（乘法都不需要），就可有 16.49% 的概率和高手的选择一模一样。

为说明这个策略网络的具体运作，我们看一个相对简单的局面，来自于 AlphaGo 与柯洁的第 2 局的第 27 手。图 4-19 中带红点的 Q15 是刚下的一手，来自于持黑的 AlphaGo。

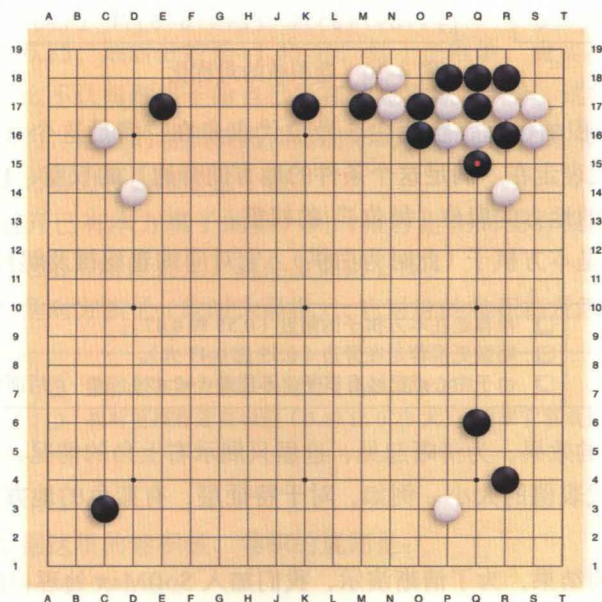


图 4-19 待分析的局面

如果读者对于围棋的基本规则有了解，此时持黑的 AlphaGo 在叫吃白方的 P17、P16、Q16 这 3 子，因为这 3 子此时只有 1 口气了，如果白置之不理，黑再走 P15 就会把它们吃掉。

弃这 3 个子的代价太大，所以持白的柯洁的下一手可以说只有 2 种选择：

- 要么走 P15，逃出 3 子。如果黑再走 O15 叫吃，则白可走 P14 成功逃出。
- 要么走 R15，把黑的 R16 吃掉，也可暂时解除威胁。但黑一定会再走 P15 叫吃，则白必须走 R16 连回，然后黑可走 O18 也把自己连上，于是白有“一团棋被封在右边”的感觉。

柯洁最终的选择是 P15。让我们看策略网络对此有何看法。

4.2.3 最简化的策略网络：特征层和卷积后的结果

如前所述，我们采用 8 个特征层，那么每个特征层有 1 个对应的 3×3 卷积核。由于这里卷积层的输出是直接输入 SoftMax，所以可忽略偏置。

由于我们训练时会随机将棋盘做 8 种对称变换。那么，对于这个简单网络，最终得到的卷积核应具有同样的对称性。但实际上，由于 SGD 有一定的随机性，因此训练出的卷积核基本对称，但并不是完全对称。这里为了方便展示，将卷积核做如图 4-20 所示的对称化。

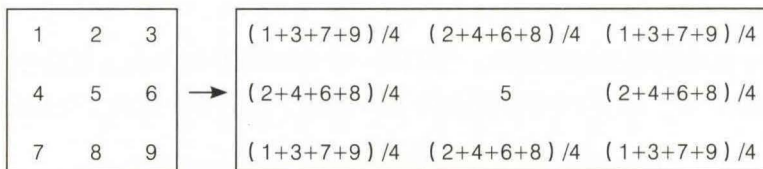


图 4-20 卷积核的对称化

如果读者思考卷积的过程，会明白 5 号位代表走在“满足这个条件的地方”的权重，而 2、4、6、8 号位代表走在“满足这个条件的地方的旁边”的权重，1、3、7、9 号位代表走在“满足这个条件的地方的肩位（斜位）”的权重。

第 1 个特征层，是本方棋子（此时为白棋）。它对应的卷积核及解释：

0.47	0.55	0.47
0.55	-2.47	0.55
0.47	0.55	0.47

- 推荐走在本方棋子的附近（0.55 和 0.47）。
- 略微更推荐走在旁边（0.55 比 0.47 大）。
- 由于中心点已经有棋子，不能走（-2.47），这一点后面也经常出现，不再赘述。

用图像说明卷积的效果，为清晰起见，这里只展示右上角的情况，如图 4-21 所示。

- 蓝色的深度代表值的大小。例如，对于特征层，有蓝色的地方代表 1，无蓝色的地方代表 0。
- 对于卷积后的结果，为了清晰演示，我们加入 SoftMax 处理，然后将最大概率选点的概率放大到 1，然后也用蓝色的深浅表示。



图 4-21 第 1 个特征层和卷积效果

进一步解释图 c 是怎么算出来的。策略网络是给每个点打分，分数越高就代表越应该走在这里。分数是通过卷积得到的，根据这里的卷积核，可得：

- 每个点的初始分数为 0 分。
- 每颗白子给自己所在的点贡献 -2.47 分。
- 每颗白子给自己上下左右邻近的 4 个点贡献 0.55 分。
- 每颗白子给自己肩位的 4 个点贡献 0.47 分。

举例：

- L15、R19 这样的点，本身不是白子，周围也没有白子，所以分数为 0。
- L19 会受到 M18（肩位）的影响，所以分数为 0.47。
- O18 会受到 N18（直接接触，贡献为 0.55）和 N17、P17（肩位接触，贡献都为 0.47）的影响，所以分数为 $0.55+0.47+0.47=1.49$ 分，分数很高，所以在 c 图的颜色很深。
- O17 的分数更高，是 $0.55+0.55+0.47+0.47=2.04$ 分。但是这个地方已经有黑子了，怎么可以走在现有的棋子上面？没关系，后面有修正的办法。
- M18 本身是白子，白子自己给自己所在的点贡献 -2.47，然后它的两个白子邻居给它贡献 $0.47+0.55$ ，最后分数是 -1.45 分，低于正常值，确实不适合走。

读者会问，为什么不让每颗棋子给自己所在的点贡献 $-\infty$ 分，这样就肯定不会走在现有的棋子上面。这是因为，这里的卷积核是训练出来的，后面会看到 -2.47 其实已足够小，网络在这种情况下会缺乏动力再去缩小这个值。

最终效果，是避开已有本方棋子的点，同时推荐走在周围有很多本方棋子的空点（如 O17、Q17、R16）。把结果略作美化，就是上面的右图。

第 2 个特征层，是对方棋子（此时为黑棋）。它对应的卷积核及解释：

0.23	0.49	0.23
0.49	-2.27	0.49
0.23	0.49	0.23
<ul style="list-style-type: none"> □ 推荐走在对方棋子的附近（0.49 和 0.23）。 □ 相对更推荐走在旁边（0.49 比 0.23 大）。看来它喜欢短兵相接。 		

效果如图 4-22 所示。

和前面一样，根据这里的卷积核，卷积的规则是：

- 每个点的初始分数为 0 分。
- 每颗黑子给自己所在的点贡献 -2.27 分。

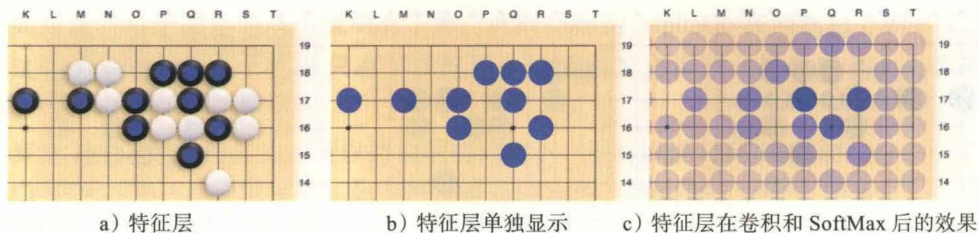


图 4-22 第 2 个特征层和卷积效果

- 每颗黑子给自己上下左右邻近的 4 个点贡献 0.49 分。
- 每颗黑子给自己肩位的 4 个点贡献 0.23 分。

例如 P17 的得分很高，因为它周围都是黑子，而且自己不是黑子。

不过这里也有问题，因为 P17 已经有白子了，是不能下的。在此说明，总共有 8 个特征层，最后会把 8 个特征层的分数加起来，加起来之后就会解决这种问题。

于是这个特征层的效果，是避开已有对方棋子的点，同时推荐走在周围有很多对方棋子的点（如 P17、R17）。

第 3 个特征层，是空点：

0.41	0.09	0.41
0.09	5.78	0.09
0.41	0.09	0.41

□ 当然应该走在空点（5.78）。

□ 空旷的空点更好（0.41 和 0.09）。

效果如图 4-23 所示。

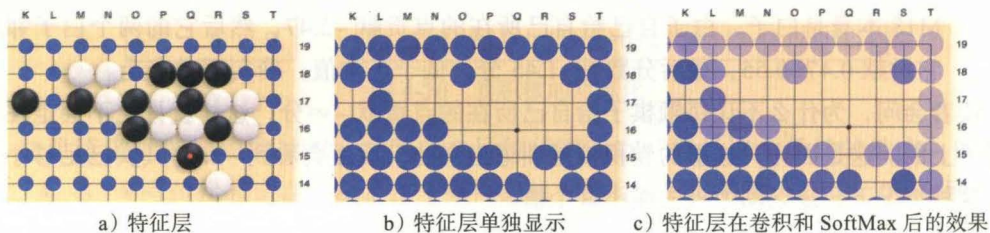


图 4-23 第 3 个特征层和卷积效果

由于空点给自己所在的位置贡献 5.78 分，因此非空点的位置就享受不到这 5.78 分，这意味着会非常倾向于选择空点。因为 SoftMax 关注的是点之间的相对分数，所以“给空点加 5.78 分”就等价于“给非空点减 5.78 分”。

值得注意的是，图 4-23c 显示的输出很有意思，我们发现，它避开有棋子的地方，避开拥挤的空点（如 P15、R15），也避开棋盘最外圈，因为最外圈周围的空点不够多。

不妨验证如下：

- L15 是空点，附近 8 个点也是空点，所以它的分数是： $5.78+0.09+0.09+0.09+0.09+0.41+0.41+0.41=7.78$ 分。

- T19 是空点，但在角落，所以附近的空点只有 S19、S18、T18。所以它的分数是： $5.78+0.09+0.09+0.41=6.37$ 分，确实比 L15 低了一些，整整低了 1.41 分。我们在后文会看到，这种“忽略棋盘外的点”的做法，实际是在使用补零外衬 (zero padding)。
- 根据 SoftMax 的定义，如果两个点的分数差 X ，选择的概率就差 e^X 倍。所以，如果差 1.41 分，选择的概率是差 $e^{1.41}=4.1$ 倍。

将前 3 个特征平面的打分相加后的结果，如图 4-24 所示。

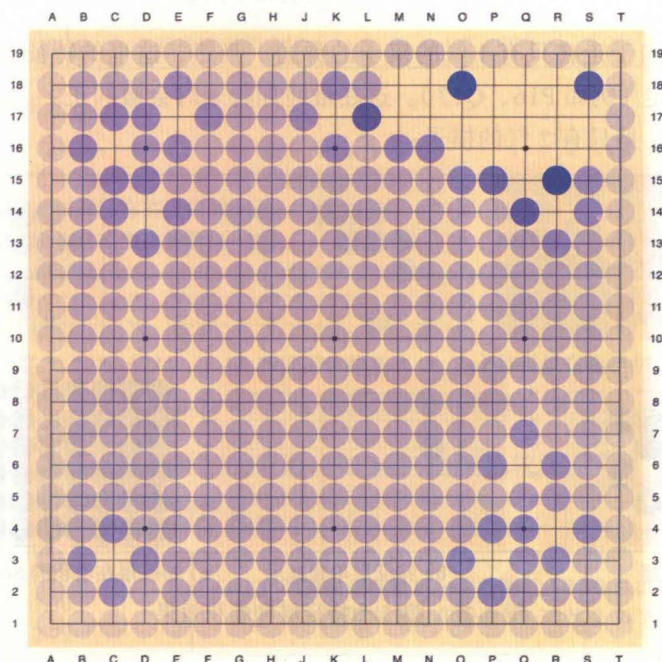


图 4-24 前 3 个特征层在卷积后的综合效果

蓝色最深的地方分数最高，可见，是 O18、R15 这样的与双方棋子都有直接接触的空点。这两个地方确实有重要性，鉴于我们的方法如此简单，确实值得赞许。

早期的围棋程序（如我国陈志行教授开发的《手谈》）与这里的思想有些相似。但当时的计算机处理能力太低，也没有这么多的棋谱数据，所以当时的开发者只能手工一个个实验各种参数，效率很低，效果也差。

接着看第 4 个特征层，是只有 1 气的棋子：

<div> <div>-0.26</div> <div>1.30</div> <div>-0.26</div> </div> <div> <div>1.30</div> <div>-0.92</div> <div>1.30</div> </div> <div> <div>-0.26</div> <div>1.30</div> <div>-0.26</div> </div>	<div> <div>□</div> <div>明显倾向于走在它们的旁边 (1.30)。</div> </div> <div> <div>○</div> <div>如果是对方的棋子，就是吃子。</div> </div> <div> <div>○</div> <div>如果是本方的棋子，就是救被打吃的棋子。</div> </div> <div> <div>□</div> <div>走在它们的肩位无意义 (-0.26)。</div> </div>
---	---

效果如图 4-25 所示。

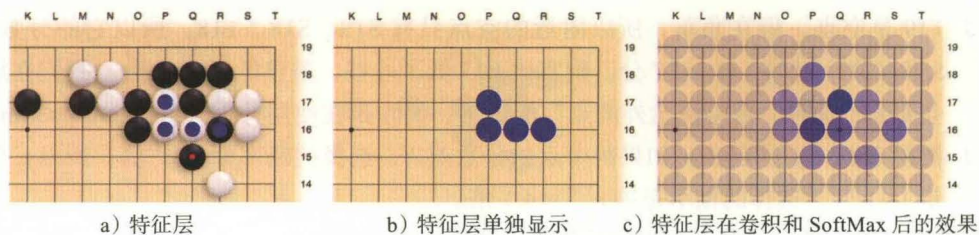


图 4-25 第 4 个特征层和卷积效果

可见，明显倾向于走在只有 1 气的棋子的旁边。当然，这里的特征层也不知道很多地方已经有子，不能走（例如 P16、Q17）。这在最终加起来时会被纠正。

第 5 个特征层，是只有 2 气的棋子：

0.42	0.21	0.42
0.21	-1.28	0.21
0.42	0.21	0.42

☐ 倾向走在它们肩位 (0.42)。

☐ 也可考虑走在旁边 (0.21)。

效果如图 4-26 所示。

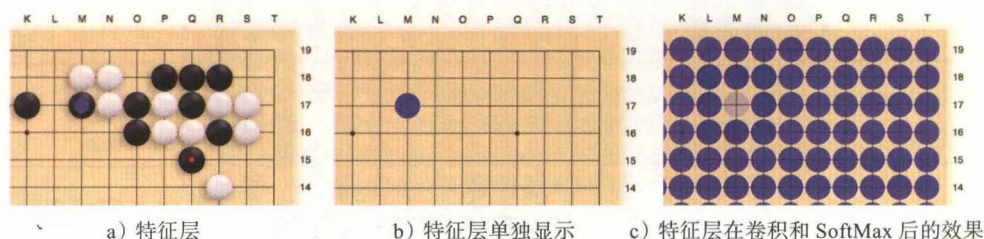


图 4-26 第 5 个特征层和卷积效果

这里最高分数的点也只有 0.42 分（是 L17、N17、M18、M16 这 4 个点），和普通点（它们的分数为 0）的差别不大，所以在图 4-26c 中看上去差别不是很明显。

第 6 个特征层，是只有 3 气的棋子：

0.41	-0.02	0.41
-0.02	-1.32	-0.02
0.41	-0.02	0.41

☐ 倾向走在它们的肩位 (0.41)。

☐ 略微避开走在旁边 (-0.02)。

效果如图 4-27 所示。



图 4-27 第 6 个特征层和卷积效果

由于 P16 的权重达到 0.80 分 ($0.41+0.41-0.02=0.80$)，与普通点的分数拉开了差距，让它们显得浅了。

第 7 个特征层，是有 4 气及以上的棋子：

0.11	-0.38	0.11
-0.38	-1.65	-0.38
0.11	-0.38	0.11

避免走在它们的旁边 (-0.38)。这有一定道理。

- 如果是本方的强势棋子，再接触就容易走凝重。
- 如果是对方的强势棋子，很多时候不适合直接接触。

略微考虑走在它们的肩位 (0.11)。

- 由于这里的卷积核只有 3×3 ，且网络只有 1 层，所以描述不了“小飞”等距离更远的走法。

效果如图 4-28 所示。

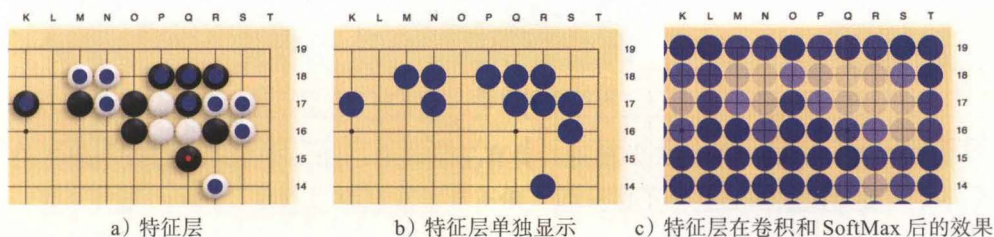


图 4-28 第 7 个特征层和卷积效果

在图中可看到“避免直接接触”的效果，如 M17 的颜色较浅。

第 8 个特征层，是最后一手的位置：

2.79	3.30	2.79
3.30	0.04	3.30
2.79	3.30	2.79

很倾向走在最后一手的附近 (3.30 和 2.79)。

相对更倾向于走在旁边 (3.30 比 2.79 大)。

效果如图 4-29 所示。



图 4-29 第 8 个特征层和卷积效果

在卷积后，基本只剩下最后一手的附近棋子。这个特征层很重要。如果不使用上一手的位置，预测准确率就会只有 4.93%。

最终，将 8 个特征平面的打分相加后的结果，是网络的最终输出，如图 4-30 所示，成功地将目标锁定在了 P15 和 R15。

从图 4-30 可见最后一手的效应非常强，其他地方的蓝色都已经看不到了。这个网络本身的棋力很低，因为它太简单，基本只会靠着最后一手的位置走。这是浅层神经网络的局限性。如果用深度神经网络，最后一手的效应会减少到正常的程度。

实际上，预测准确率需达到 30% 才开始有初学者的棋力，达到 50% 就下得颇为不错。后文我们训练的策略网络可达到 50% 以上的预测准确率。

如果去掉最后一手的特征层（第 8 个特征层），结果如图 4-31 所示。

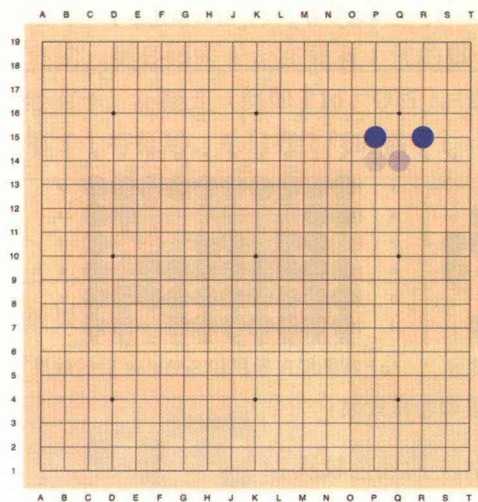


图 4-30 网络的最终输出

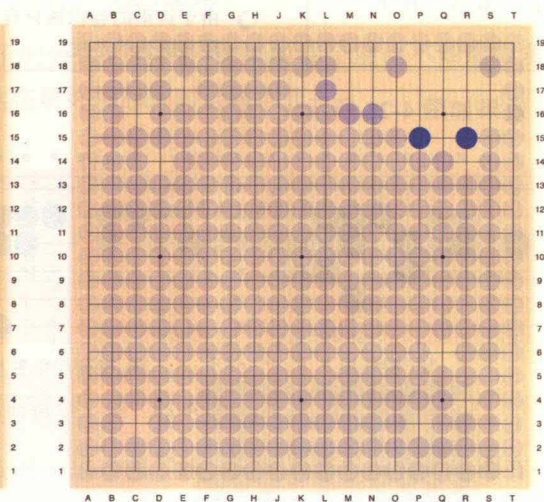


图 4-31 去掉第 8 个特征层后的效果

可见此时最关键的是“走在只有 1 气的棋子的旁边”。读者可将这里的过程写成代码，实验更多的局面来看效果。

由于这里的卷积核只有 3×3 ，而且只有 1 层网络，所以这个网络也会喜欢走在现有棋子的旁边。通过使用深度神经网络也会解决这个问题。

4.3 卷积神经网络：进一步了解

4.3.1 卷积核、滤波器与参数量的计算

对于卷积的另一种理解方法，是将它看成是经典图像处理中的滤波器（filter）。例如，在 Mathematica 执行下列代码：

```
img = ExampleData[{"TestImage", "House"}] (* 载入示例图像 *)
ImageConvolve[img, ({
  {-1, 0, 1},
  {-1, 0, 1},
  {-1, 0, 1}
}]] (* 卷积操作，这个滤波器可找到图像中垂直的边缘 *)
ImageConvolve[img, ({
```



```
{1/9, 1/9, 1/9},
{1/9, 1/9, 1/9},
{1/9, 1/9, 1/9}
}]] (* 卷积操作, 这个滤波器可将图像模糊 *)
```

将会看到如图 4-32 所示的输出。



图 4-32 图像滤波效果

在卷积操作后，图像中的值往往会有正有负，正值代表与特征匹配，负值代表与特征相反。如果再进行 ReLU 操作，就会只留下正值，即只留下与特征匹配的位置。因此 ReLU 很适合与卷积配合。

最后，我们用计算说明卷积网络在参数数量上的优势。考虑下列输入和输出：

- 输入：3 张 32×32 的图像（代表红、绿、蓝 3 个通道）。
- 输出：64 张 30×30 的图像（经过 3×3 卷积核，图像尺寸缩小 2 个像素）。

如果使用卷积网络：

- 卷积核：如前所述，如果希望将 A 张图变成 B 张图，那么需要 $A \times B$ 个卷积核。
 - 因此，这里需要 $3 \times 64 = 192$ 个卷积核。
 - 卷积核是 3×3 ，则有 9 个参数。
 - 另外每个输出图像还有 1 个偏置，那么 64 个输出图像就有 64 个偏置。
- 参数量： $192 \times 9 + 64 = 1792$ 。

如果使用全连接网络：

- 神经元：由于输出是 $64 \times 30 \times 30 = 57\,600$ 个像素，因此需要 57600 个神经元。
 - 每个神经元有 $3 \times 32 \times 32 = 3072$ 个输入权重，加上偏置有 3073 个参数。
- 参数量： $57\,600 \times 3073 = 177\,004\,800$ ，远远大于卷积网络的参数量。可以想象，运行会非常慢，而且由于参数的数量极多，容易出现过拟合。

4.3.2 运作和训练的计算

为了检验自己是否掌握了卷积神经网络，最好的办法是计算一遍。考虑一个简单的卷积神经网络，如图 4-33 所示。

这里的“全局最大池化”（global max pooling）会在后文讲述，其实就是输出图像的所

有点的最大值。

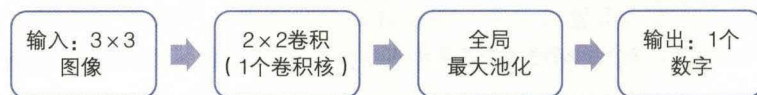


图 4-33 待分析的卷积网络

前向传播时， 3×3 图像经过 2×2 卷积会变为 2×2 图像，再经过全局最大池化就变为 1 个数字。令图像的像素值为 $\{x_{11}, x_{12}, \dots, x_{33}\}$ ，卷积核的像素值为 $\{w_{11}, w_{12}, w_{21}, w_{22}\}$ ，则具体公式如下：

$$O_{11} = w_{11}x_{11} + w_{12}x_{12} + w_{21}x_{21} + w_{22}x_{22} + b$$

$$O_{12} = w_{11}x_{12} + w_{12}x_{13} + w_{21}x_{22} + w_{22}x_{23} + b$$

$$O_{21} = w_{11}x_{21} + w_{12}x_{22} + w_{21}x_{31} + w_{22}x_{32} + b$$

$$O_{22} = w_{11}x_{22} + w_{12}x_{23} + w_{21}x_{32} + w_{22}x_{33} + b$$

$$\text{OUT} = \text{MAX}(O_{11}, O_{12}, O_{21}, O_{22})$$

$$\text{LOSS} = (\text{OUT} - \text{期望输出})^2$$

再看反向传播，首先：

$$\frac{\partial \text{LOSS}}{\partial \text{OUT}} = 2(\text{OUT} - \text{期望输出})$$

然后 MAX 函数的求导方法如下：

$$\frac{\partial \text{OUT}}{\partial O_{ij}} = \begin{cases} 1 & \text{若 } O_{ij} \text{ 为 } O_{11}, O_{12}, O_{21}, O_{22} \text{ 中最大者} \\ 0 & \text{若 } O_{ij} \text{ 非 } O_{11}, O_{12}, O_{21}, O_{22} \text{ 中最大者} \end{cases}$$

再计算出 $\frac{\partial O_{ij}}{\partial w_{kl}}$ ，即可算出最终的梯度，例如：

$$\frac{\partial \text{LOSS}}{\partial w_{12}} = \frac{\partial \text{LOSS}}{\partial \text{OUT}} \cdot \frac{\partial \text{OUT}}{\partial w_{12}} = 2(\text{OUT} - \text{期望输出}) \cdot \begin{cases} x_{12} & \text{若 } O_{11} \text{ 为 } O_{11}, O_{12}, O_{21}, O_{22} \text{ 中最大者} \\ x_{13} & \text{若 } O_{12} \text{ 为 } O_{11}, O_{12}, O_{21}, O_{22} \text{ 中最大者} \\ x_{22} & \text{若 } O_{21} \text{ 为 } O_{11}, O_{12}, O_{21}, O_{22} \text{ 中最大者} \\ x_{23} & \text{若 } O_{22} \text{ 为 } O_{11}, O_{12}, O_{21}, O_{22} \text{ 中最大者} \end{cases}$$

细心的读者会问，如果有多于 1 个的 O_{ij} 恰好都是最大，该如何处理。由于 O_{ij} 是实数，这种事情发生的概率近乎为 0，如果恰好遇上，在其中任选 1 个即可。

总之，只要按部就班写出每一项，卷积神经网络的反向传播就并不困难。对于更复杂的卷积神经网络，也可用类似的方法计算，还可进一步写成矩阵的形式以加速计算（方法称为 im2col，感兴趣的读者可自行搜索）。

4.3.3 外衬与步长

如前所述，在卷积后，图像会缩小。但我们常常希望图像尺寸不变，原因如下：

□ 如果图像在卷积后尺寸不变，就可不断叠加卷积层，网络可以更深，效果往往更好。例如，对于 AlphaGo 中的网络，我们希望 19×19 棋盘图像在卷积后仍然维持在 19×19 的大小，这样可使用很多层的卷积网络。

□ 如果需使用后文会提到的残差 (residue) 结构，那么也会希望图像在卷积后维持原尺寸。

为此，可在卷积前，给图像加入补零外衬 (zero padding)，即用 0 在上下左右的外边缘填充 p 个像素，让待卷积的图像变得更大，如图 4-34 所示。

由图可见，如设置外衬为 1 像素，即可让图像在 3×3 卷积后维持原尺寸。

一般而言，如果设置外衬为 p 像素，则 $n \times n$ 图像在外衬后为 $(n+2p) \times (n+2p)$ ，如果再用 $(2p+1) \times (2p+1)$ 的滤波器作用，就可得到仍然是 $n \times n$ 的图像。

这就是为什么我们往往会使用大小为 3×3 、 5×5 等奇数大小的卷积核，因为此时可通过设置合适的外衬，使得图像在卷积后大小不变。

最后，我们还可使用非 0 的值作为外衬，如采用其他固定值、重复边缘点、在边缘做镜像等方法，它们在 MXNet 中均可通过调用 `mx.sym.Pad()` 实现。

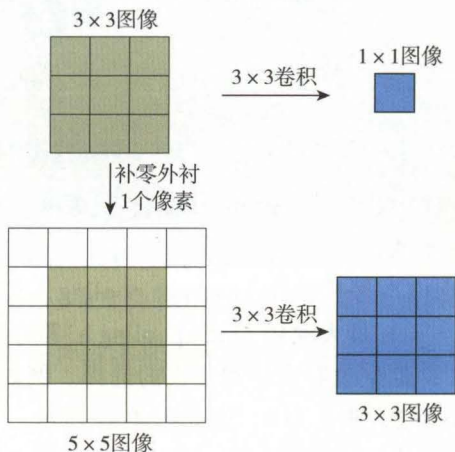


图 4-34 补零外衬的效应

在另一些时候，我们希望图像在卷积后缩小得

更快，例如直接变为原来的一半大小。一种办法是使用卷积的步长 (stride)，另一种办法是使用后文会提到的池化层。

卷积的步长，就是卷积核每步所移动的距离，可用例子说明。

之前我们做的卷积，都是步长为 1 的卷积，即卷积核每次移动 1 个像素。如图 4-35 所示， 4×4 的图像，经过大小为 3，步长为 1 的卷积，得到的是 2×2 的图像。

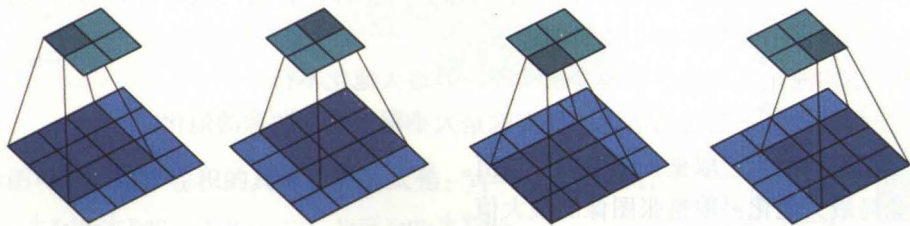


图 4-35 步长为 1 的卷积

下面看步长为 2 的例子，如图 4-36 所示，即卷积核每次移动 2 个像素。可见， 5×5 的图像，经过大小为 3，步长为 2 的卷积，得到的是 2×2 的图像。

一般而言，边长为 n 的图像，经过大小为 k ，步长为 s 的卷积，得到的是边长为 $\frac{n-k}{s} + 1$ 的图像。当不能整除时 (即卷积核有部分超出图像范围时)，不同框架的取整方法不同，值

得读者注意。目前 MXNet 的规范是向下取整，读者可通过加入外衬或调整输入图像大小，微调尺寸。

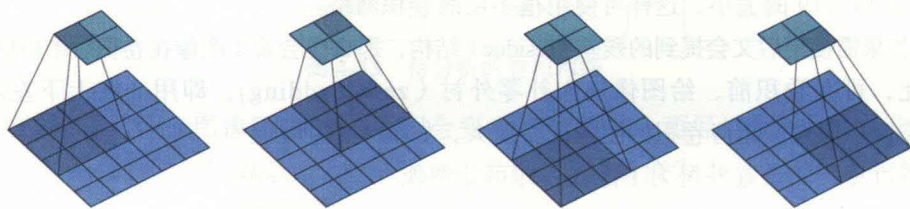


图 4-36 步长为 2 的卷积

4.3.4 缩小图像：池化与全局池化

池化层 (pooling layer)，是深度卷积网络中另一种快速缩小图像的方法。常用的池化方法有两种：

- ❑ 平均池化 (average pooling，又称为 mean pooling)：取小范围内的平均值。
- ❑ 最大池化 (max pooling)：取小范围内的最大值。

池化也有大小和步长的选择。例如，若大小为 2，步长为 2，则如图 4-37 所示。

在图像处理中，最大池化较为常用。举例，假设我们需要找的特征是小狗，那么只要图像中有一个区域存在小狗，就说明整张图像存在小狗，所以我们应取图像中所有区域与小狗特征的匹配度的最大值，因此用最大池化更适合。

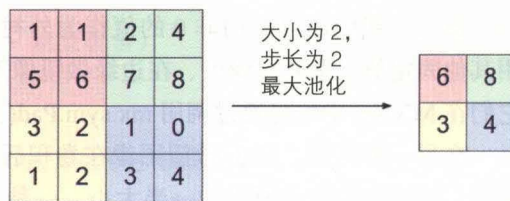


图 4-37 最大池化

还可实验一个小技巧：3×3 池化大小，2×2 步长。这来自于后文会介绍的著名 AlexNet 架构，可让池化结果更平滑，有时效果会更好。

池化与带步长的卷积，都能快速缩小图像。具体哪种方法的效果好，需由实验决定。有的网络会同时使用这两种缩小图像的方法。

注意，平均池化可用带步长的卷积实现，但最大池化不行。

另一个相关的概念是全局池化，其实就是大小覆盖整张图像的池化：

- ❑ 全局平均池化：取整张图像的平均值。
- ❑ 全局最大池化：取整张图像的最大值。

全局池化很适合放在网络的末端。例如，如果我们希望把图像分为 10 类，那么可先通过不断卷积和池化，得到 10 张小图，再做全局池化，得到 10 个数字，再加上 SoftMax 层就可作为输出的 10 个概率。

全局池化的一大优势是无任何参数，有时还可直接显示网络的判断依据。例如，对于后文会介绍的 CIFAR-10 分类问题，效果如图 4-38 所示。

一般而言，如果 A 个像素经过某种卷积会变成 B 个像素，那么 B 个像素经过相应的转置卷积会变成 A 个像素。

值得注意，如果仔细分析转置卷积的算法，会发现它生成的图像可能出现棋盘状图案，如图 4-41 所示。

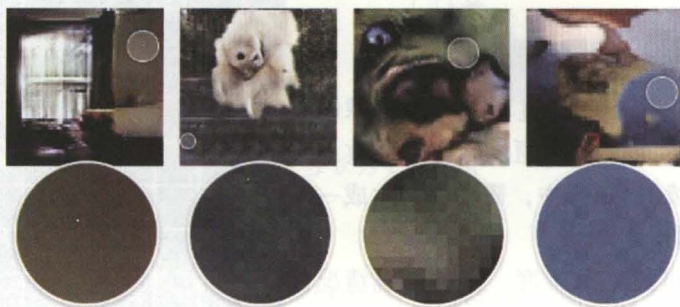


图 4-41 转置卷积的棋盘图案

在《Deconvolution and Checkerboard Artifacts》^①一文对此有详细的分析。文中指出，这个现象在步长不能整除卷积大小的时候会最明显。例如，如果读者画出大小为 3，步长为 2 的转置卷积，会发现计算公式中每个像素的项目数不同，这就是该现象的成因所在。

如果读者遇到了这种现象，可让步长能整除卷积大小，或采用《Is the deconvolution layer the same as a convolutional layer?》^②一文的方法，先生成 nr^2 个 $K \times K$ 通道，然后将通道的像素循环排序，即可得到 n 个 $rK \times rK$ 通道。

如图 4-42 所示，生成 r^2 个通道后，按照图中用彩色标注的次序将像素重新排序，即可得到 1 个边长为 r 倍的通道。

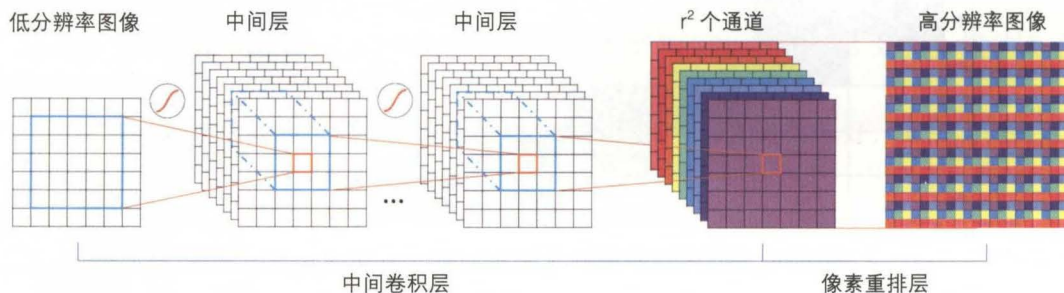


图 4-42

4.4 实例：用卷积网络解决 MNIST 问题

在此，我们训练一个可在 MNIST 问题上达到 99.4% 识别准确率的卷积网络。

① 地址为 <https://distill.pub/2016/deconv-checkerboard/>。

② 地址为 <https://arxiv.org/abs/1609.07009>。

4.4.1 网络架构的定义与参数的计算

本小节用到了批规范化 (Batch Normalization, BN 或 BatchNorm) 层, 会在后文介绍。BN 层可加速网络的收敛, 常常被放置在线性激活层之前, 甚至可放在每个非线性激活层之前。

这也说明现在的深度学习框架很容易使用, 因为我们还没有介绍 BN 的工作原理, 但读者也可轻松地使用, 深度学习框架会负责背后的所有计算。

网络的定义代码:

```
data = mx.symbol.Variable('data')

# 第1个卷积层, 以及相应的BN和非线性, 有32个5×5卷积
conv1 = mx.sym.Convolution(data=data, name="conv1",
kernel=(5,5), num_filter=32)
# 对于BN层我们往往会加上fix_gamma=False这个参数
bn1 = mx.sym.BatchNorm(data=conv1, name="bn1", fix_gamma=False)
act1 = mx.sym.Activation(data=bn1, name="act1", act_type="relu")
# 第1个池化层, 使用了3×3大小和2×2步长
pool1 = mx.sym.Pooling(data=act1, name="pool1",
pool_type="max", kernel=(3,3), stride=(2,2))

# 第2个卷积层, 以及相应的BN和非线性, 有64个5×5卷积
conv2 = mx.sym.Convolution(data=pool1, name="conv2",
kernel=(5,5), num_filter=64)
bn2 = mx.sym.BatchNorm(data=conv2, name="bn2", fix_gamma=False)
act2 = mx.sym.Activation(data=bn2, name="act2", act_type="relu")
# 第2个池化层, 使用了3×3大小和2×2步长
pool2 = mx.sym.Pooling(data=act2, name="pool2",
pool_type="max", kernel=(3,3), stride=(2,2))

# 第3个卷积层, 有10个3×3卷积
conv3 = mx.sym.Convolution(data=pool2, name="conv3",
kernel=(3,3), num_filter=10)
# 第3个池化层, 这里设置global_pool进行全局池化, 会忽略kernel的值
pool3 = mx.sym.Pooling(data=conv3, name="pool3",
global_pool=True, pool_type="avg", kernel=(1,1))
# 将图像摊平, 这里的效果是将10张1×1的图像摊平, 因此得到10个数
flatten = mx.sym.Flatten(data=pool3, name="flatten")
# SoftMax层, 将10个数变为10个分类的概率
net = mx.sym.SoftmaxOutput(data=flatten, name='softmax')
```

网络图代码如下, 其效果图如图 4-43 所示。

```
shape = {"data": (batch_size, 1, 28, 28)}
mx.viz.plot_network(symbol=net, shape=shape).view()
```

由网络图可见, 这里的最后 1 个全局池化层实际没有

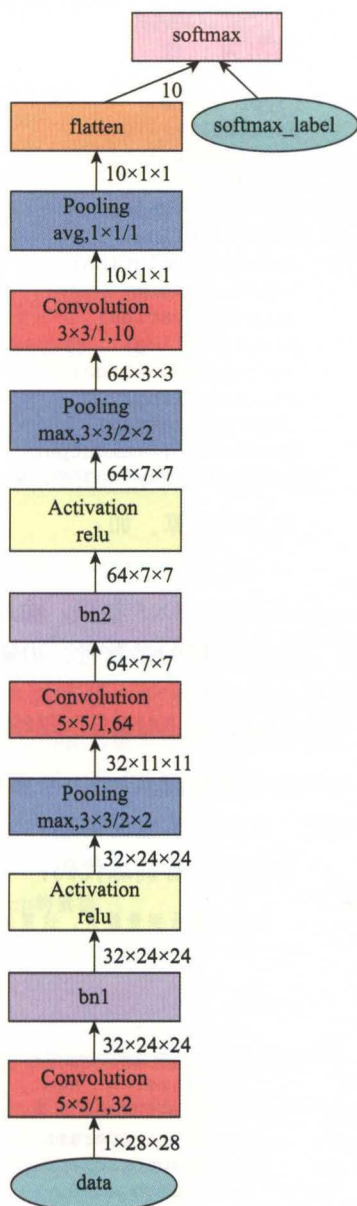


图 4-43 网络结构图

作用，它是为了方便读者做实验。例如，如果我们把全局池化前的卷积核大小改为 1×1 ，那么全局池化就会起作用，但最终效果不如用 3×3 卷积核。可认为最后 1 个卷积层类似于鉴别缩小的数字，所以用 3×3 卷积核区分的效果更好。

读者可验算其中的图像尺寸，例如 $(24-3)/2+1$ 向下取整等于 11。

下面看参数量：

```
mx.viz.print_summary(symbol=net, shape=shape)
```

输出为：

Layer (type)	Output Shape	Param #	Previous Layer
data(null)	1x28x28	0	
conv1(Convolution)	32x24x24	832	data
bn1(BatchNorm)	32x24x24	64	conv1
act1(Activation)	32x24x24	0	bn1
pool1(Pooling)	32x11x11	0	act1
conv2(Convolution)	64x7x7	51264	pool1
bn2(BatchNorm)	64x7x7	128	conv2
act2(Activation)	64x7x7	0	bn2
pool2(Pooling)	64x3x3	0	act2
conv3(Convolution)	10x1x1	5770	pool2
pool3(Pooling)	10x1x1	0	conv3
flatten(Flatten)	10	0	pool3
softmax(SoftmaxOutput)	10	0	flatten
Total params: 58058			

读者可验算，如：

- 32 个 5×5 卷积，输入是 1 个通道，参数量 $= 32 \times (1 \times 5 \times 5 + 1) = 32 \times 26 = 832$ 。
- 64 个 5×5 卷积，输入是 32 个通道，参数量 $= 64 \times (32 \times 5 \times 5 + 1) = 64 \times 801 = 51264$ 。
- BN 层的参数量，为输入通道数 $\times 2$ ，后文会解释。

4.4.2 训练 MNIST 网络

载入库和读入 MNIST 数据的代码，与第 3 章全连接网络的代码完全相同。定义网络的代码，在前一节也已讲述。

在此只需看训练代码：

```
# 由于训练数据量较大，这里采用了GPU，若读者没有GPU，可修改为CPU
module = mx.mod.Module(symbol=net, context=mx.gpu(0))

module.fit(
    train_iter,
    eval_data=val_iter,
    optimizer = 'sgd',
    # 采用0.03的初始学习速率，并在每60000个样本后（即每1个epoch后）将学习速率缩减为之前的0.9倍
    optimizer_params = {'learning_rate' : 0.03, 'lr_scheduler' : mx.lr_scheduler.
        FactorScheduler(step=60000/batch_size, factor=0.9)},
    num_epoch = 20,
    batch_end_callback = mx.callback.Speedometer(batch_size, 60000/batch_size)
)
```

运行输出如下, 首先 MXNet 会确定当前 GPU 上最快的卷积算法:

```
[21:27:49] d:\program files (x86)\jenkins\workspace\mxnet\mxnet\src\operator\./
  cudnn_algoreg-inl.h:112: Running performance tests to find the best
  convolution algorithm, this can take a while... (setting env variable MXNET_
  CUDNN_AUTOTUNE_DEFAULT to 0 to disable)
```

然后训练过程如下:

```
INFO:root:Epoch[0] Train-accuracy=0.973633
INFO:root:Epoch[0] Time cost=6.736
INFO:root:Epoch[0] Validation-accuracy=0.987320
INFO:root:Update[1876]: Change learning rate to 2.70000e-02
INFO:root:Epoch[1] Train-accuracy=0.990183
INFO:root:Epoch[1] Time cost=6.738
INFO:root:Epoch[1] Validation-accuracy=0.990016
INFO:root:Update[3751]: Change learning rate to 2.43000e-02
.....
INFO:root:Epoch[18] Train-accuracy=0.999933
INFO:root:Epoch[18] Time cost=6.864
INFO:root:Epoch[18] Validation-accuracy=0.994010
INFO:root:Update[35626]: Change learning rate to 4.05256e-03
INFO:root:Epoch[19] Train-accuracy=0.999950
INFO:root:Epoch[19] Time cost=6.840
INFO:root:Epoch[19] Validation-accuracy=0.994010
```

最终在训练集的准确率为 99.995%, 在测试集的准确率为 99.40%。这个性能已经很好, 如果希望继续提高, 可加入数据增强。

4.4.3 在 MXNet 运行训练后的网络

在训练完网络后, 下一步就是运行网络。这里的代码可放在前文的训练代码后执行。

首先, 可通过 DataIter 提供数据。注意, 在定义 module 并训练后, 它的 batch_size 已经被绑定。因此在运行网络时, 也需每次送入 batch_size 个数据, 除非重新改变绑定方法。

举例, 由于之前定义的 val_iter 是每次送入 batch_size 个数据, 所以无需改变 module 的绑定, 可直接用 val_iter 提供数据:

```
val_iter.reset() # 将val_iter重置, 保证是从头开始提供数据
# 前向传播, 用next()进行val_iter的迭代, 每次调用可得到一个batch的数据
module.forward(val_iter.next(), is_train=False)
out = module.get_outputs()[0].asnumpy() # 得到输出
# 将输出做一些美观处理
print(zip(out.argmax(axis=1), out.max(axis=1)))
```

输出为识别结果和对结果的信心。例如, 第 1 张图像被认为有 0.9999988 的几率为数字 7:

```
[(7, 0.9999988), (2, 0.99998844), (1, 0.99991703), (0, 0.99999046), (4,
0.99999738), (1, 0.99993443), (4, 0.9999485), (9, 0.99999893), (5,
0.93937075), (9, 0.99996662), (0, 0.99998677), (6, 0.99999893), (9,
0.99988949), (0, 0.99999952), (1, 0.999982), (5, 0.99995184), (9, 1.0),
(7, 0.99999881), (3, 0.97729844), (4, 0.9999994), (9, 0.99997473), (6,
0.99994755), (6, 0.99999857), (5, 1.0), (4, 0.99997091), (0, 0.99999487),
```



```
(7, 0.99997842), (4, 1.0), (0, 0.99999869), (1,
0.99982411), (3, 1.0), (1, 0.99959022)]
```

不妨人工检验识别结果是否准确，以第 1 张图像为例：

```
plt.imshow(val_img[0].reshape(28,28), cmap='Greys_r')
plt.axis('off')
plt.show()
print(val_lbl[0])
```

这张图像确实为数字 7，如图 4-44 所示。

我们再找前 15 个 batch 中是否有识别错误的例子：

```
val_iter.reset()
for i in range(0,15):
    # 如前所述，每次调用val_iter.next()都会得到一个batch的数据
    module.forward(val_iter.next(), is_train=False)
    out = module.get_outputs()[0].asnumpy().argmax(axis=1)
    for j in range(0,len(out)):
        if out[j] != val_lbl[i*batch_size + j]: # 是否错误识别？
            print(out[j], val_lbl[j]) # 输出错误的情况
            plt.imshow(val_img[j].reshape(28,28), cmap='Greys_r')
            plt.axis('off')
            plt.show()
```



图 4-44 MNIST 的第 1 张图像

通常会发现几个，例如，将图 4-45 中的 9 识别为 4：

这个数字确实潦草，但人类不会看错，因此网络仍然存在提高空间。

下面再看如何手工将数据送入网络（即，不使用 DataIter），以及如何改变网络的绑定方式。

```
# 定义一个简单数据结构，方便后续 module 读取数据
from collections import namedtuple
MyBatch = namedtuple('MyBatch', ['data', 'label'])
```



图 4-45 MNIST 中识别错误的例子

```
# 对于不熟悉 namedtuple 的朋友，这里举例说明 namedtuple 的运作方式：
# 如果此时执行 MyBatchExample = MyBatch('abc', 'def')
# 那么 print(MyBatchExample.data) 就会输出 'abc'
# 而 print(MyBatchExample.label) 就会输出 'def'
```

```
# 将 module 重新绑定为批大小为1
new_batch_size = 1
```

```
module.bind(data_shapes=[('data', (new_batch_size, 1, 28, 28))], label_
shapes=[('softmax_label', (new_batch_size,))], force_rebind=True, for_
training=False) # 设置force_rebind为True，说明是在重新绑定；设置for_training为
False，说明只作前向传播
```

```
# 这里的数据使用val_img的第1张。我们也可送入自己生成的数据
# 这里将MyBatch的第2个参数设为None，因为它对应标签，而运行网络不需要标签
MyBatchData=MyBatch([mx.nd.array(val_img[0].reshape(1,1,28,28).astype(np.float 32))],None)
```

```
# 运行网络，module 会读取 MyBatchData.data 和 MyBatchData.label
module.forward(MyBatchData)
```

```
# 关闭 print 的科学计数法显示, 让结果更美观
np.set_printoptions(suppress=True)
# 输出网络运行结果
print(module.get_outputs()[0].asnumpy())
```

输出为图像属于数字 0 到 9 的概率:

```
[[ 0.          0.          0.00000004  0.00000006  0.          0.          0.
  0.99999988  0.          0.00000016]]
```

可见这张图像 (val_img 的第 1 张) 的最高概率所属为数字 7, 概率为 0.99999988, 与之前的结果一致。

4.4.4 调参实例

在此我们主要看批大小。学习速率维持此前的设定 (即, 初始值为 0.03 且每个 epoch 降低到原来的 0.9 倍)。训练长度维持为 20 个 epoch。

- ❑ 若批大小设置为 32, 在笔者的机器上, 训练速度为每个 epoch 在 6.8 秒左右, 最终可达到 99.40% 的训练精度。
- ❑ 若设置为 128, 训练速度为每个 epoch 在 2.4 秒左右, 最终可达到 99.33% 的训练精度。速度更快, 精度略低。
- ❑ 若设置为 512, 训练速度为每个 epoch 在 1.4 秒左右, 最终可达到 99.10% 的训练精度。速度更快, 精度降低较多。
- ❑ 若设置为 8, 训练速度为每个 epoch 在 21.8 秒左右, 最终可达到 99.37% 的训练精度。速度很慢, 精度相对接近。

可见, 批大小越大, 训练速度往往越快, 但训练精度会受到影响。为了提高训练精度, 可尝试同时提高学习速率, 但网络的训练过程会容易不稳定。

例如, 若将批大小设置为 512, 将初始学习速率提高到 0.2, 则有时可达到 99.3% 的训练精度, 有时达不到, 每次的结果变化较大。

因此, 选取一个合理的兼具训练速度与训练效果的批大小, 十分重要。在前文我们提过 Facebook 对于很大的批大小的训练有一定研究, 读者可参考并实验。

4.4.5 在 Fashion-MNIST 数据集的结果

使用与前文一致的代码:

```
INFO:root:Epoch[0] Train-accuracy=0.829933
INFO:root:Epoch[0] Time cost=6.545
INFO:root:Epoch[0] Validation-accuracy=0.860923
INFO:root:Update[1876]: Change learning rate to 2.70000e-02
INFO:root:Epoch[1] Train-accuracy=0.883317
INFO:root:Epoch[1] Time cost=6.503
INFO:root:Epoch[1] Validation-accuracy=0.869708
.....
INFO:root:Epoch[19] Train-accuracy=0.968450
```

```
INFO:root:Epoch[19] Time cost=6.566
INFO:root:Epoch[19] Validation-accuracy=0.916833
```

再将第 1 和第 2 卷积层的隐神经元数分别加大为 64 和 128。学习速率增大为 0.06:

```
INFO:root:Epoch[0] Train-accuracy=0.810650
INFO:root:Epoch[0] Time cost=7.828
INFO:root:Epoch[0] Validation-accuracy=0.868311
INFO:root:Update[1876]: Change learning rate to 5.40000e-02
INFO:root:Epoch[1] Train-accuracy=0.881400
INFO:root:Epoch[1] Time cost=7.843
INFO:root:Epoch[1] Validation-accuracy=0.889177
.....
INFO:root:Epoch[19] Train-accuracy=0.988233
INFO:root:Epoch[19] Time cost=7.846
INFO:root:Epoch[19] Validation-accuracy=0.919129
```

可见在训练集的准确率明显提高，达到 98.8%，但在测试集的准确率提高很少，仍然停留在 91.9%，说明单纯加大网络容易导致过拟合。我们在后文会用更先进的网络架构，并配合数据增强，提高性能。

4.5 MXNet 的使用技巧

4.5.1 快速定义多个层

对于复杂的网络结构，如果手工定义每一层，费时费力，且容易出错。在此看几个技巧。

1) 可缩写常用的层描述。如定义：

```
BatchNorm = mx.sym.BatchNorm
```

那么，对于下面的语句：

```
bn1 = mx.sym.BatchNorm(data=conv1, name="bn1", fix_gamma=False)
```

可简化为：

```
bn1 = BatchNorm(data=conv1, name="bn1", fix_gamma=False)
```

2) 可将常用层用函数描述。例如再定义：

```
def Conv_BN_Act(src, layer, kernel, num_filter):
    conv = mx.sym.Convolution(data=src, name="conv"+layer, kernel=kernel, num_
filter=num_filter)
    bn = BatchNorm(data=conv, name="bn"+layer, fix_gamma=False)
    act = mx.sym.Activation(data=bn, name="act"+layer, act_type="relu")
    return act
```

那么，对于下面的段落：

```
conv1 = mx.sym.Convolution(data=data, name="conv1", kernel=(5,5), num_filter=64)
bn1 = BatchNorm(data=conv1, name="bn1", fix_gamma=False)
act1 = mx.sym.Activation(data=bn1, name="act1", act_type="relu")
```


可简化为一句:

```
layer1 = Conv_BN_Act(data, layer="1", kernel=(5,5), num_filter = 64)
```

3) 可将层自动串联。例如:

```
data = mx.symbol.Variable('data')
for i in range(0, 3):
    net = Conv_BN_Act(data if i==0 else net, str(i), (3,3), 128)
```

就等价于:

```
data = mx.symbol.Variable('data')
net = Conv_BN_Act(data, "0", (3,3), 128)
net = Conv_BN_Act(net, "1", (3,3), 128)
net = Conv_BN_Act(net, "2", (3,3), 128)
```

也等价于:

```
data = mx.symbol.Variable('data')
net0 = Conv_BN_Act(data, "0", (3,3), 128)
net1 = Conv_BN_Act(net0, "1", (3,3), 128)
net2 = Conv_BN_Act(net1, "2", (3,3), 128)
```

这是因为中间层的变量名只是为了描述网络, 实际在 MXNet 储存的是每一层中的 name 变量。因此, 如果读者希望省事, 可在中间层使用相同的变量名。

4.5.2 网络的保存与读取

完整的训练程序, 需能将网络的参数保存下来, 用于继续训练和后续使用。

若我们定义的 module 名为 mod, 则可通过下列语句保存网络参数:

```
mod.save_params(prefix)
```

其中 prefix 是可以自定义的文件名前缀。调用后, 将在代码所在目录, 保存以 params 为后缀的文件, 其中是网络的参数。

而读取参数, 可通过在网络初始化时采用下列语句:

```
mod.init_params(initializer=mx.init.Load(prefix))
```

此外, 我们有时需要保存和读取网络的架构。保存可通过:

```
mod._symbol.save('%s-symbol.json' % prefix)
```

将得到 JSON 格式的文本文件, 文件中是网络的定义数据, 包括每层的定义和每层之间的连接方法。

读取可通过:

```
symbol = mx.sym.load('%s-symbol.json' % prefix)
```

4.5.3 图像数据的打包和载入

之前我们通过 NDArryIter 向 MXNet 提供数据。但如需处理大量数据, 就应手动定义

数据迭代器，采用多线程异步载入，这样 MXNet 就无需频繁等待数据。

不过，这样的代码也较为复杂。为此，MXNet 提供了图像打包格式 RecordIO，以及专用迭代器 ImageRecordIter，它可快速异步载入打包后的图像，还可自动完成剪切、缩放、图像增强等常见操作。在此我们先看打包图像的方法。

1) 将图像文件按分类标签放置在相应的子目录中，例如：

- ❑ 在 C:/DeepLearning/img/0 存放标签为 0 的图像。
- ❑ 在 C:/DeepLearning/img/1 存放标签为 1 的图像。
- ❑ 在 C:/DeepLearning/img/2 存放标签为 2 的图像。

以此类推。

2) 下载 MXNet 的 GitHub 项目，地址为 <https://github.com/dmlc/mxnet>。

3) 在命令行进入 tools 子目录，生成图像列表 lst 文件。例如对于 c:/DeepLearning/img 下的图像，执行：

```
python im2rec.py --list=1 --recursive=1 c:/DeepLearning/img c:/DeepLearning/img
```

注意，如果在 Windows 下运行，有可能会因为以文本方式打开图像而读取失败，可修改 im2rec.py 中的 open 函数加入 "rb" 参数代表以二进制方式读取图像。

4) 从列表生成 rec 文件和 idx 文件：

```
python im2rec.py --pass-through=1 c:/DeepLearning c:/DeepLearning/img
```

5) 将生成的 lst 和 rec 和 idx 文件移动到代码所在目录，即可在 ImageRecordIter 载入，例如：

```
train_iter = mx.io.ImageRecordIter(
    path_imgrec = "train.rec", # 此前生成的rec文件名
    data_shape = (3,32,32), # 数据格式，3*32*32代表彩色32*32图像
    batch_size = batch_size, # 批大小
    shuffle = True, # 开启乱序
)
```

4.5.4 深入 MXNet 训练细节

在前文我们是直接调用 module.fit() 实现训练，但有时我们需要对于 MXNet 的训练过程有更精细的控制（例如在后文训练 GAN 的时候），因此本节将讲述如何使用 MXNet 的更底层函数实现训练。

不妨以前几章的简单找大小神经网络为例，比较 MXNet 的计算是否与我们的计算一致。首先是常规的定义：

```
##--coding:utf-8--

import logging
import math
import random
```

```

import mxnet as mx # 导入 MXNet 库
import numpy as np # 导入 NumPy 库, 这是 Python 常用的科学计算库

n_sample = 2 # 训练用的样本个数
batch_size = 1 # 批大小
learning_rate = 0.1 # 学习速率
n_epoch = 1 # 训练 epoch 数

train_in = [[0.5, 1], [0.2, 0.6]] # 为测试, 设置2个固定样本
train_out = [1, 0.6] # 设置相应的训练目标

# 为测试, 关闭了随机顺序
train_iter = mx.io.NDArrayIter(data = np.array(train_in), label = {'reg_label': np.array(train_out)}, batch_size = batch_size, shuffle = False)

src = mx.sym.Variable('data') # 输入层
fc1 = mx.sym.FullyConnected(data = src, num_hidden = 2, name = 'fc1') # 全连接层
act1 = mx.sym.Activation(data = fc1, act_type = "relu", name = 'act1') # ReLU层
fc2 = mx.sym.FullyConnected(data = act1, num_hidden = 1, name = 'fc2') # 全连接层
net = mx.sym.LinearRegressionOutput(data = fc2, name = 'reg') # 输出层

```

手动定义模组和初始化:

```

# 定义模组
mod = mx.mod.Module(symbol = net, label_names = (['reg_label']))
# 需手动绑定训练数据和标签
mod.bind(data_shapes=train_iter.provide_data, label_shapes=train_iter.provide_label)
# 为测试, 采用纯手动初始化:
mod.init_params(arg_params={
    'fc1_weight':mx.nd.array([[0.5, 0], [0.5, 1]]),
    'fc1_bias':mx.nd.array([0, 0]),
    'fc2_weight':mx.nd.array([[0.5, 0.5]]),
    'fc2_bias':mx.nd.array([0])
})
# 正常情况下, 可用MXNet的自带函数初始化, 例如:
# mod.init_params(initializer=mx.init.Uniform(scale=0.1))

# 设置优化器
mod.init_optimizer(optimizer='sgd', optimizer_params=({'learning_rate', 0.1},))
# 设置需跟踪的性能指标
metric = mx.metric.create('mse')

```

手动训练:

```

for epoch in range(1): # 手动训练, 这里只训练1个epoch
    train_iter.reset() # 每个epoch需手动将迭代器复位
    # 实际训练时, 应在此调用 metric.reset() 将性能指标复位
    for batch in train_iter: # 对于每个batch...
        print('===== input =====')
        print(batch.data) # 数据
        print(batch.label) # 数据的标签
        mod.forward(batch, is_train=True) # 前向传播
        print('===== output =====')
        print(mod.get_outputs()) # 网络的输出
        metric.reset() # 这里希望看网络的训练细节, 所以对于每个样本都将指标复位

```



```

mod.update_metric(metric, batch.label) # 更新指标
print('===== metric =====')
print(metric.get()) # 指标的情况
mod.backward() # 反向传播, 计算梯度
print('===== grads =====')
print(mod._exec_group.grad_arrays) # 输出梯度情况
mod.update() # 根据梯度情况, 由优化器更新网络参数
print('===== params =====')
print(mod.get_params()) # 输出新的参数
print('\n')

```

程序运行输出如下(代码中有 2 个样本, 下面是第 1 个样本的输出):

```

===== input =====
[
[[ 0.5  1. ]]
<NDArray 1x2 @cpu(0)>]
[
[ 1.]
<NDArray 1 @cpu(0)>]
===== output =====
[[ 0.75]]
<NDArray 1x1 @cpu(0)>]
===== metric =====
('mse', 0.0625)
===== grads =====
[[
[[-0.0625 -0.125 ]
 [-0.0625 -0.125 ]]
<NDArray 2x2 @cpu(0)>], [
[-0.125 -0.125]
<NDArray 2 @cpu(0)>], [
[[-0.0625 -0.3125]]
<NDArray 1x2 @cpu(0)>], [
[-0.25]
<NDArray 1 @cpu(0)>]]
===== params =====
({'fc2_bias':
[ 0.025]
<NDArray 1 @cpu(0)>, 'fc2_weight':
[[ 0.50625002  0.53125   ]]
<NDArray 1x2 @cpu(0)>, 'fc1_bias':
[ 0.0125  0.0125]
<NDArray 2 @cpu(0)>, 'fc1_weight':
[[ 0.50625002  0.0125   ]
 [ 0.50625002  1.01250005]]
<NDArray 2x2 @cpu(0)>}, {})

```

如果读者与前文对照, 会发现这里的梯度(就是损失对于参数的偏导)和最终的参数变化情况, 都只有第 2 章的计算值的 1/2。这是因为 MXNet 在训练时使用的 MSE 定义是 $(X-Y)^2/2$, 多了一个 1/2 因子。在考虑这个因素后, 就和第 2 章的计算完全一致。

4.5.5 在浏览器和移动设备运行神经网络

训练神经网络后，下一步就是部署。如果神经网络只能在安装了深度学习框架的电脑上运行，那就显得太局限了。

为此，已有多种库可在不同的设备和环境中快速运行神经网络，例如：

- 对于浏览器，可使用 WebDNN、deeplearn.js、Keras.js、TensorFire 等。具体而言会通过 WebAssembly、asm.js、WebGL、WebGPU 等技术。
- 对于电脑，可使用 nVidia 的 TensorRT，DMLC 的 NNVM 等等。
- 对于移动设备，可使用 iOS 的 CoreML、高通的 NPE、ARM 的 ComputeLibrary、百度的 mobile-deep-learning、腾讯的 ncnn 等。

通过这些方法运行神经网络，有可能比在深度学习框架中更快，因为它们有可能会进一步分析网络架构，编译最优化的代码。

目前快速运行 MXNet 模型的方法，是将 MXNet 模型转换为这些库所需的格式。这需要少量工作，或将 MXNet 转化为其他框架的模型。在 <https://github.com/ysh329/deep-learning-model-converto> 有不同框架间互相转化模型的方法，而 ONNX 计划 (<http://onnx.ai/>) 也在推进框架间模型的转换。

MXNet 还提供了 amalgamation 技术，可将神经网络的运行代码整合为一个巨大的 C 文件，由于 C 文件可再编译到多种语言，最终可让神经网络在浏览器的 JavaScript 中直接运行，或在 iOS 和 Android 设备上直接运行，缺点是速度较慢。读者可在 MXNet 的 GitHub 主页中搜索 amalgamation，查看具体介绍。

以浏览器为例：

- 1) 在 <https://github.com/dmlc/mxnet.js/> 下载编译后的 amalgamation 代码。
- 2) 将其中的 mxnet_predict.js、libmxnet_predict.js、libmxnet_predict.js.mem 保存到网页所在的目录中。

3) 用 tools 子目录下的 model2json.py 将储存的 MXNet 模型转换为 JSON 格式。

4) 在网页 JavaScript 中引入 mxnet_predict.js 和 libmxnet_predict.js。载入模型：

```
$.getJSON("model/policy.json", function(model) { // 载入JSON格式模型
    policyNetwork = new Predictor(model, {
        'data': [1, 8, 19, 19] // 设置数据输入格式
    });
});
```

5) 令 input_data 为网络输入，则运行网络的方法为：

```
policyNetwork.setInput('data', ndarray(input_data, [1, 8, 19, 19]));
policyNetwork.forward();
console.log(policyNetwork.output(0).data) // 输出
```

下面再看在 iOS 的 CoreML 库中的运行方法^①。可用其中的 mxnet-to-coreml 工具将 MXNet

① 介绍见 <https://github.com/apache/incubator-mxnet/tree/master/tools/coreml>。

模型转换为 CoreML 所需的格式，然后参照 <https://developer.apple.com/documentation/coreml> 中的介绍，在 iOS 载入和运行，如图 4-46 所示。

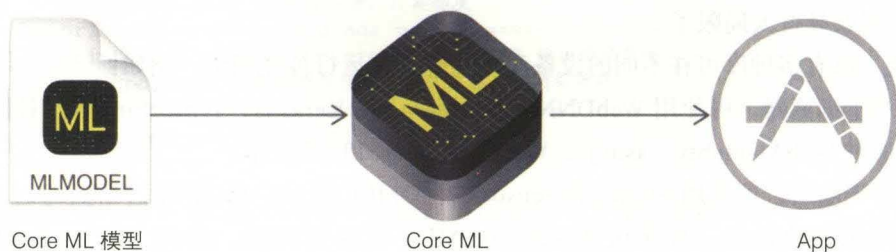


图 4-46 使用 CoreML 运行网络

最后，如果读者的应用对于速度有较高要求，甚至可自行编写深度网络的运行代码，这个过程其实并不复杂。

简单的方法是使用 BLAS 库中的 `sgemm` 函数做矩阵乘法，且图像卷积操作需通过 `im2col` 操作写成矩阵形式。对此感兴趣的读者，可查看笔者在 GitHub 的开源项目《BlinkDL》(<https://github.com/BlinkDL/BlinkDL>)，其中包括 JavaScript 实现的深度卷积网络运行代码，且代码只有 200 多行。

而如果读者的编程能力较强，熟悉各种算法，更快速的方法是使用 FFT（快速傅立叶变换）以及 Winograd 算法，可参见 NNPACK 项目[⊖]。

如需再使用 GPU 加速，可参阅 cudnn、MIOpen、OpenCL、WebGL、WebGPU 的文档。

[⊖] 地址为 <https://github.com/Maratyszczka/NNPACK>。

深度卷积网络：第四课

深度卷积网络，通过使用多层结构，显著提高了网络的性能和泛化能力。经典的例子是在 ImageNet 2012 图像识别竞赛上夺冠的 AlexNet，它让诸多业界人士首次看到了深度学习的巨大威力和潜力，标志着深度学习革命的开启。

这里的 ImageNet 是由美国 Stanford 大学李飞飞教授和 Princeton 大学李凯教授发起的图像标记数据库，目前共有 1400 多万张图像，其主页是 <http://www.image-net.org/>。

自 2010 年以来，每年会举办 ImageNet 图像识别竞赛，其中经典的问题是图像分类和画框（Object classification + localization），即识别出图像中主体的类别，并画出主体的包围框（bounding box），如图 5-1 所示。

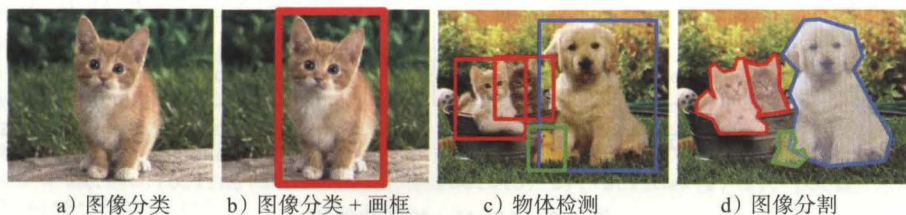


图 5-1 经典的图像问题

ImageNet 竞赛的训练数据共有 120 余万张图像，分为 1000 个类别，如：狮子狗、绒猴、晴雨表、创可贴、萨克斯等，其中还特意包括了上百种犬类品种，用于考验模型的辨别能力。

请看历年的 ImageNet 赛果，本章的重点是其中的图像分类问题（注意，表中的分类错误率，是指模型的前 5 位预测不包括正确答案的概率，可称为 Top5 错误率），如表 5-1 所示。

表 5-1 ImageNet 赛果

年份	2010	2011	2012	2013	2014	2015	2016	2017
分类错误率	28.19%	25.77%	15.32%	11.20%	7.40%	3.57%	2.99%	2.25%
画框误差	未举办此项	0.425	0.335	0.299	0.253	0.090	0.077	0.062
著名获胜者			AlexNet		VGG、Inception	ResNet		

表中标出了图像分类问题上的著名获胜者，可见 AlexNet 带来了图像分类准确率的首次飞跃，影响深远，在它之前很少有神经网络模型参赛，而在它之后的参赛模型几乎都是深度卷积网络。

在 AlexNet 之后的著名图像分类获胜者包括 VGG、Inception 和 ResNet，我们也会在后文介绍。

5.1 经典的深度卷积网络架构

5.1.1 深度学习革命的揭幕者：AlexNet

AlexNet 的原始论文[⊖]来自 Hinton 带领的多伦多大学团队。由于当时的 GPU 能力有限，其中使用了双 GPU 训练，且将网络也相应切成两部分，两部分只在部分层之间沟通，如图 5-2 所示。

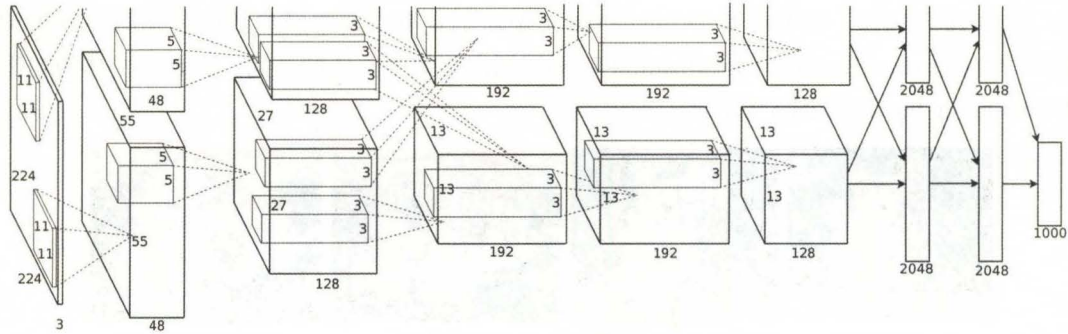


图 5-2 AlexNet 的原始架构

随着 GPU 的发展，我们已经可用单 GPU 轻松训练 AlexNet，因此可将 AlexNet 的两部分合并如图 5-3 所示（注意，这会令连接更多，运算量更大）。

注意，在卷积核有部分超出图像范围时，不同框架的取整方法不同，因此不同资料的中间层的图像大小可能有差异。由于 MXNet 是向下取整，所以得到的图像会略小一些，在下面的代码中会看到。

⊖ 地址为 <http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf>。

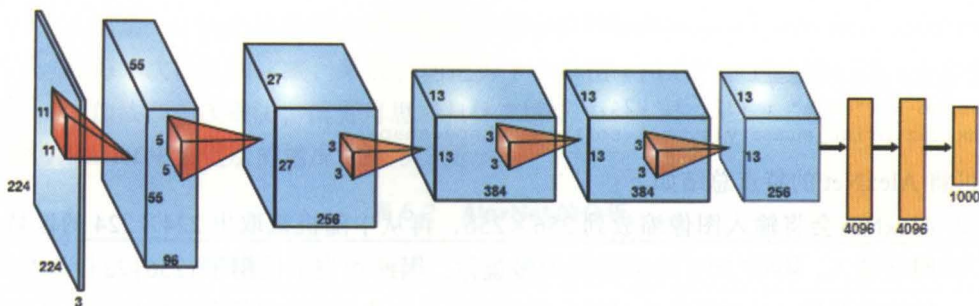


图 5-3 AlexNet 的合并架构

在 MXNet 中定义 AlexNet 方法，以及每层对于通道和图像尺寸的变换情况，如下：

```
data = mx.symbol.Variable('data')
# 将3*224*224变换为96*54*54
conv1 = mx.sym.Convolution(name='conv1', data=data, kernel=(11, 11), stride=(4, 4), num_filter=96)
relu1 = mx.sym.Activation(data=conv1, act_type="relu")
# 将96*54*54变换为96*26*26 (最大池化)
pool1 = mx.sym.Pooling(data=relu1, pool_type="max", kernel=(3, 3), stride=(2,2))
# 将96*26*26变换为256*26*26
conv2 = mx.sym.Convolution(name='conv2', data=pool1, kernel=(5, 5), pad=(2, 2), num_filter=256)
relu2 = mx.sym.Activation(data=conv2, act_type="relu")
# 将256*26*26变换为256*12*12 (最大池化)
pool2 = mx.sym.Pooling(data=relu2, kernel=(3, 3), stride=(2, 2), pool_type="max")
# 将256*12*12变换为384*12*12
conv3 = mx.sym.Convolution(name='conv3', data=pool2, kernel=(3, 3), pad=(1, 1), num_filter=384)
relu3 = mx.sym.Activation(data=conv3, act_type="relu")
# 将384*12*12变换为384*12*12
conv4 = mx.sym.Convolution(name='conv4', data=relu3, kernel=(3, 3), pad=(1, 1), num_filter=384)
relu4 = mx.sym.Activation(data=conv4, act_type="relu")
# 将384*12*12变换为256*12*12
conv5 = mx.sym.Convolution(name='conv5', data=relu4, kernel=(3, 3), pad=(1, 1), num_filter=256)
relu5 = mx.sym.Activation(data=conv5, act_type="relu")
# 将256*12*12变换为256*5*5 (最大池化)
pool3 = mx.sym.Pooling(data=relu5, kernel=(3, 3), stride=(2, 2), pool_type="max")
# 将256*5*5变换为6400
flatten = mx.sym.Flatten(data=pool3)
# 将6400变换为4096
fc1 = mx.sym.FullyConnected(name='fc1', data=flatten, num_hidden=4096)
relu6 = mx.sym.Activation(data=fc1, act_type="relu")
dropout1 = mx.sym.Dropout(data=relu6, p=0.5)
# 将4096变换为4096
fc2 = mx.sym.FullyConnected(name='fc2', data=dropout1, num_hidden=4096)
relu7 = mx.sym.Activation(data=fc2, act_type="relu")
dropout2 = mx.sym.Dropout(data=relu7, p=0.5)
# 将4096变换为1000
fc3 = mx.sym.FullyConnected(name='fc3', data=dropout2, num_hidden=1000)
```



```
softmax = mx.sym.SoftmaxOutput(data=fc3, name='softmax')
```

输出参数情况供参考

```
shape = {"data" : (32, 3, 224, 224)}
```

```
mx.viz.print_summary(symbol=softmax, shape=shape)
```

可将 AlexNet 的特点总结如下：

- ❑ AlexNet 会将输入图像缩放到 256×256 ，再从中随机截取出 224×224 的区域作为网络输入，并有 50% 几率将图像做镜像，因此相当于使用了 $(256-224)^2 \times 2 = 2048$ 倍的数据量。这对于防止过拟合至关重要，也是目前常用的图像数据增强思路。
 - 在运行 AlexNet 时，也可取图像的 4 角加中心共 5 个位置，再配合镜像，把 1 张图像变成 10 张图像，取 10 个预测的平均值，可提高少许性能。
 - AlexNet 的原始论文还会给图像加少量低频噪声，进一步增强数据。
- ❑ 由于是彩色图像，因此输入有 3 个通道，可写成 $3 \times 224 \times 224$ 。经 5 次卷积和 2 次最大池化，变为 $256 \times 12 \times 12$ ，即 256 个 12×12 图像。
 - AlexNet 中的非线性激活都采用 ReLU。
 - AlexNet 中的最大池化均为大小 3，步长 2。
 - AlexNet 中的大部分卷积使用了外衬，以使卷积后图像尺寸不变。
 - 在原始 AlexNet 中还会加入 Local Response Normalization (LRN) 层，用于将 ReLU 的激活略微归一化，可略微提高性能，不过作用不明显，因此后续模型并没有再使用。现代方法是使用 BatchNorm 层。
- ❑ 再经 1 次最大池化，变为 $256 \times 5 \times 5$ ，摊平后变为 6400 个数字，经 3 个全连接层，变为 1000 个数字，再经 SoftMax 即为对于 1000 个类别的分类结果。
 - 在网络最后使用多个全连接层是传统的做法，稍后我们会看到这会造成参数量大大增加。现代方法是尽量采用纯卷积的结构，可显著减少参数量。
 - 为减少过拟合，AlexNet 在全连接层之间使用了 Dropout，每次移除 50% 神经元，这是 AlexNet 所发明的技巧。
- ❑ 对于卷积后图像的大小，不妨做个验算：第 1 个卷积层的卷积核为 11×11 ，步长为 4，因此得到的图像的边长为 $(224-11)/4+1$ ，向下取整后为 54。注意 AlexNet 使用了高达 11×11 的卷积核。现代方法是只使用 3×3 卷积核，配合更深的网络。

最后我们计算每层的参数量、数据内存消耗、计算量 (FLOPs)。举例，第 1 个卷积层，输入为 $3 \times 224 \times 224$ ，输出为 $96 \times 54 \times 54$ ，卷积大小为 11，因此：

- ❑ 参数量为 $96 \times (1+3 \times 11 \times 11) = 34944$ 个。
- ❑ 数据内存消耗为 $4 \times 96 \times 54 \times 54 = 1119744$ 字节 ≈ 1.12 MB。这里假设为 32bit 浮点数，每个占用 4 字节。
- ❑ 计算量约为 $2 \times 96 \times 54 \times 54 \times 3 \times 11 \times 11 = 203233536 \approx 203$ M。

- 这是一个粗略的估算，我们关心的是它的数量级。
- 每1次浮点加、减、乘法，均记为1个单位（FLOP）。这里乘以2，来自于1次加法和1次乘法，读者可思考具体来源。

则 AlexNet 中关键层的情况如表 5-2 所示。

表 5-2 AlexNet 的分析

层名	参数量	占比	数据内存消耗	计算量	占比
输入			0.60 MB		
conv1	34 944	0.1%	1.12 MB	203 M	10.1%
conv2	614 656	1.2%	0.69 MB	831 M	41.1%
conv3	885 120	1.7%	0.22 MB	255 M	12.6%
conv4	1 327 488	2.6%	0.22 MB	382 M	18.9%
conv5	884 992	1.7%	0.15 MB	255 M	12.6%
fc1	26 218 496	51.6%	0.02 MB	52 M	2.6%
fc2	16 781 312	33.0%	0.02 MB	34 M	1.7%
fc3	4 097 000	8.1%	0.00 MB	8 M	0.4%
共计	50.8 M 个		3.04 MB	2.02 G	

可见，参数量大部分在全连接层，而计算量大部分在卷积层。此外，由于全连接层的参数量大，存在内存瓶颈，因此全连接层的实际速度会比这里的计算量所显示得慢一些。

5.1.2 常用架构：VGG 系列

VGG 是 AlexNet 的重要改进。在 VGG 的论文[⊖]中有几种网络架构，其中常用的是 16 层的 VGG-16，即论文中的 D 模型，如图 5-4 所示。

表中的 convK-N 代表 N 个 $K \times K$ 卷积，maxpool 代表最大池化，FC-N 代表 N 个全连接神经元，所有卷积层后有 ReLU 层。

可将 VGG-16 的架构绘制如图 5-5 所示。

VGG 系列的特点如下：

- 模型 A、B、D、E 中卷积核的大小均为 3×3 ，步长为 1，同时加入 1 个像素外衬，于是卷积后的图像尺寸不变，因此可以不断叠加 3×3 卷积，网络可以非常深。模型 C 中还使用了 1×1 卷积，这是一个重要技巧，后文会继续讨论。
- 由最大池化层负责缩小图像，大小和步长都为 2。表中有 5 个池化层，经过池化层后的图像大小分别为 112、56、28、14、7。

⊖ 地址为 <http://arxiv.org/pdf/1409.1556>。

- 随着图像的缩小，通道数增多，这有助于保留更多信息。
- 非线性全部使用 ReLU。

卷积网络架构					
A	A-LRN	B	C	D	E
11层	11层	13层	16层	16层	19层
输入 (224×224的彩色图像)					
conv3-64	conv3-64 LRN	conv3-64	conv3-64	conv3-64	conv3-64
maxpool					
conv3-128	conv3-128	conv3-128	conv3-128	conv3-128	conv3-128
maxpool					
conv3-256	conv3-256	conv3-256	conv3-256	conv3-256	conv3-256
conv3-256	conv3-256	conv3-256	conv3-256	conv3-256	conv3-256
maxpool					
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
maxpool					
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

图 5-4 VGG 论文中的架构

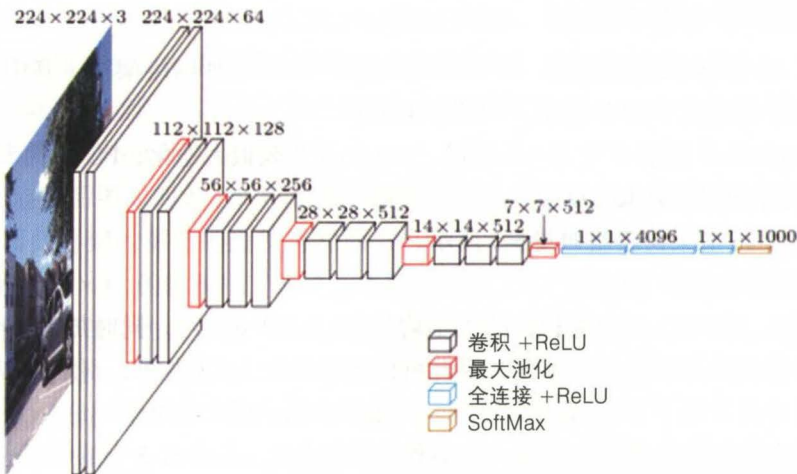


图 5-5 VGG-16 架构

VGG-16 中各层的情况如表 5-3 所示。

表 5-3 VGG-16 的分析

层名	参数量	占比	数据内存消耗	计算量	占比
输入			0.60 MB		
conv1_1	1792	0.0%	12.85 MB	173 M	0.6%
conv1_2	36928	0.0%	12.85 MB	3699 M	12.0%
conv2_1	73856	0.1%	6.42 MB	1850 M	6.0%
conv2_2	147584	0.1%	6.42 MB	3699 M	12.0%
conv3_1	295168	0.2%	3.21 MB	1850 M	6.0%
conv3_2	590080	0.4%	3.21 MB	3699 M	12.0%
conv3_3	590080	0.4%	3.21 MB	3699 M	12.0%
conv4_1	1180160	0.9%	1.61 MB	1850 M	6.0%
conv4_2	2359808	1.7%	1.61 MB	3699 M	12.0%
conv4_3	2359808	1.7%	1.61 MB	3699 M	12.0%
conv5_1	2359808	1.7%	0.40 MB	925 M	3.0%
conv5_2	2359808	1.7%	0.40 MB	925 M	3.0%
conv5_3	2359808	1.7%	0.40 MB	925 M	3.0%
fc1	102764544	74.3%	0.02 MB	206 M	0.7%
fc2	16781312	12.1%	0.02 MB	34 M	0.1%
fc3	4097000	3.0%	0.00 MB	8 M	0.0%
共计	138 M 个		54.82 MB	30.9 G	

可见，VGG 系列的参数量和计算量比 AlexNet 多得多，实际运行速度比 AlexNet 慢许多。目前最新的网络架构，可在 VGG 的几十分之一的参数量下达到相同的性能，后文会介绍。

5.1.3 去掉全连接层：DarkNet 系列

通过使用现代的架构技巧，可实现比 AlexNet 和 VGG 更快更准的深度卷积网络。例子是 <https://pjreddie.com/darknet/imagenet/> 的 DarkNet 系列。请看表 5-4 所示。

表 5-4 网络架构的性能比较

名称	Top1 正确率	Top5 正确率	计算量	Titan X 用时	i7 4GHz 用时	参数文件大小
AlexNet	57.0	80.3	2.27 G	1.5 ms	0.3 s	285 MB
DarkNet Reference	61.1	83.0	0.81 G	1.5 ms	0.16 s	28 MB
VGG-16	70.5	90.0	30.94 G	10.7 ms	4.9 s	528 MB
DarkNet-19	72.5	91.2	5.58 G	6.0 ms	0.66 s	80 MB
ResNet-50	72.9	92.9	10 G	7.0 ms	未测试	87 MB
DenseNet-201	76.4	93.7	10.9 G	未测试	未测试	66 MB

说明如下：

- DarkNet Reference 和 DarkNet-19 分别比 AlexNet 和 VGG-16 更快更准。
- ResNet 和 DenseNet 使用了残差结构，正确率更高。DarkNet 系列没有使用残差结构，运行速度更快。
- 表中的 AlexNet 比之前的 MXNet 模型 AlexNet 的图像尺寸略大，所以计算量和参数更多。
- Top1 正确率代表直接命中正确类别，Top5 正确率代表在前 5 位预测中命中正确类别。这里的输入图像只经过 1 次截取，没有取多次截取后的平均值，所以会比 ImageNet 竞赛中的正确率低一些。
- 表中可见 GPU 比 CPU 运行网络的速度约快 100 倍。
- 参数文件大小约等于参数量乘以 4 字节，因为浮点数是 4 字节。

其中 DarkNet Reference 的架构如表 5-5 所示。

表 5-5 DarkNet Reference 的分析

层名	输出通道数	输出尺寸	卷积尺寸	参数量	计算量
输入	3	224×224			
conv1	16	224×224	3×3	448	43 M
conv2	32	112×112	3×3	4640	116 M
conv3	64	56×56	3×3	18496	116 M
conv4	128	28×28	3×3	73856	116 M
conv5	256	14×14	3×3	295168	116 M
conv6	512	7×7	3×3	1180160	116 M
conv7	1024	4×4	3×3	4719616	151 M
conv8	1000	4×4	1×1	1025000	33 M
			汇总	7.3 M 个	0.81 G

最后将 conv8 输出的 1000 个 4×4 图像，经全局平均池化和 SoftMax，直接变为 1000 个概率。

其特点如下：

- 卷积层：
 - 所有 3×3 卷积都带 1 个像素外衬，因此不改变图像尺寸。
 - 所有非线性都使用 Leaky ReLU（即当 $x < 0$ 时输出 $0.1 \times x$ ）。
 - 除去最后的 conv8 外，在卷积后都有 BN 层。
 - 在 conv1 到 conv6 卷积后用最大池化缩小图像，大小和步长都为 2。在 conv6 后的最大池化加 1 个像素外衬，使结果是 4×4。
 - 架构很简洁，且由于使用纯卷积架构，参数量和计算量都大大减少。
- 再看 DarkNet-19 的架构如表 5-6 所示。

表 5-6 DarkNet-19 的分析

层名	输出通道数	输出尺寸	卷积尺寸	参数量	计算量
输入	3	224×224			
conv1	32	224×224	3×3	896	87 M
conv2	64	112×112	3×3	18 496	462 M
conv3_1	128	56×56	3×3	73 856	462 M
conv3_2	64	56×56	1×1	8 256	51 M
conv3_3	128	56×56	3×3	73 856	462 M
conv4_1	256	28×28	3×3	295 168	462 M
conv4_2	128	28×28	1×1	32 896	51 M
conv4_3	256	28×28	3×3	295 168	462 M
conv5_1	512	14×14	3×3	1 180 160	462 M
conv5_2	256	14×14	1×1	131 328	51 M
conv5_3	512	14×14	3×3	118 0160	462 M
conv5_4	256	14×14	1×1	131 328	51 M
conv5_5	512	14×14	3×3	1 180 160	462 M
conv6_1	1024	7×7	3×3	4 719 616	462 M
conv6_2	512	7×7	1×1	524 800	51 M
conv6_3	1024	7×7	3×3	4 719 616	462 M
conv6_4	512	7×7	1×1	524 800	51 M
conv6_5	1024	7×7	3×3	4 719 616	462 M
conv7	1000	7×7	1×1	1 025 000	100 M
			汇总	20.8 M 个	5.58 G

最后将 conv7 输出的 1000 个 7×7 图像，经全局平均池化和 SoftMax，直接变为 1000 个概率。

其特点如下：

□ 卷积层：

- 所有 3×3 卷积都带 1 个像素外衬，因此不改变图像尺寸。
- 所有非线性都使用 Leaky ReLU（即当 $x < 0$ 时输出 $0.1 \times x$ ）。
- 除去最后的 conv7 外，在卷积后都有 BN 层。

□ 在 conv1, conv2, conv3_3, conv4_3, conv5_5 后用最大池化缩小图像，大小和步长都为 2。

□ 灵活使用 1×1 卷积，后文会进一步介绍这一技巧：

- 在 conv3_2、conv4_2、conv5_2、conv5_4、conv6_2、conv6_4 用 1×1 卷积作为瓶颈层。
- 在 conv7 用 1×1 卷积将通道数变为所需的 1000 个。

5.2 网络的可视化：以 AlexNet 为例

由于深度卷积网络的架构复杂，参数量巨大，许多文章会声称它就像一个神秘的黑匣子。但事实可能并非如此。研究人员已有许多手段探查深度卷积网络的具体运作，并了解其决策的原因。在此我们以 AlexNet 为例。

1) 对于某张输入图像，可观察每层的神经元的输出图像（经 ReLU 后的）。以 AlexNet 的 conv1 层和 conv5 层为例，它们分别有 64 个和 256 个神经元，如图 5-6 所示。

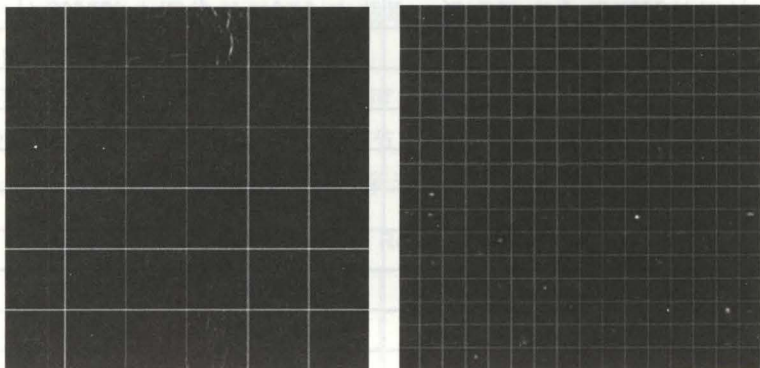


图 5-6 神经元的输出图像

可见，大多数神经元的输出是全黑的，即纯零。用专业术语说，它们具有稀疏性（sparsity）。这是件好事，因为如果希望网络准确判断图像的分类，最佳的情况就是每个神经元对应某种特征，且每次只有与图像内容真正相关的神经元被激活，其余神经元不会产生干扰。

2) 看卷积核的权重。初始的权重是随机噪声，随着网络的训练，权重将出现各种结构。可选择 conv1 和 conv2 的部分卷积核显示如图 5-7 所示。

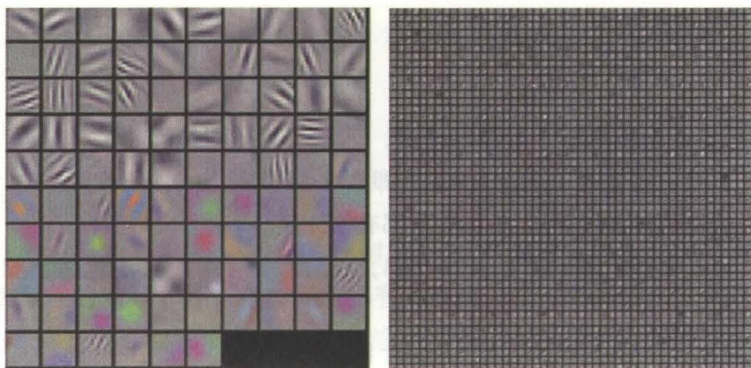


图 5-7 卷积核的权重

可见 conv1 作为第 1 层，卷积核很有特点。这里的权重来自于原始的 AlexNet，其中把

网络切开成 2 部分，因此可看到其中一半学会了高频的灰度结构，一半学会了低频的彩色特征。这里的灰度卷积核有些像计算机视觉理论中的 Gabor 滤波器，是有效的分析图像纹理的手段。

而 conv2 的卷积核是 5×5 ，比较小，就很难直接看出其目的。由于我们在现代网络中大量使用 3×3 卷积核，会更难看清楚。实际上，应该将每层的卷积核再结合之前层的卷积核，才能看到它所对应的图像特征。这就是下面方法的思路。

3) 在《Visualizing and Understanding Convolutional Networks》^①中提出了一种有效手段，对于某个神经元，可看到在训练图像中最能激活这个神经元的图像特征（可认为是神经元的识别目标），以及相应的训练图像。以 AlexNet 第 1~3 层的部分神经元为例，如图 5-8 所示。

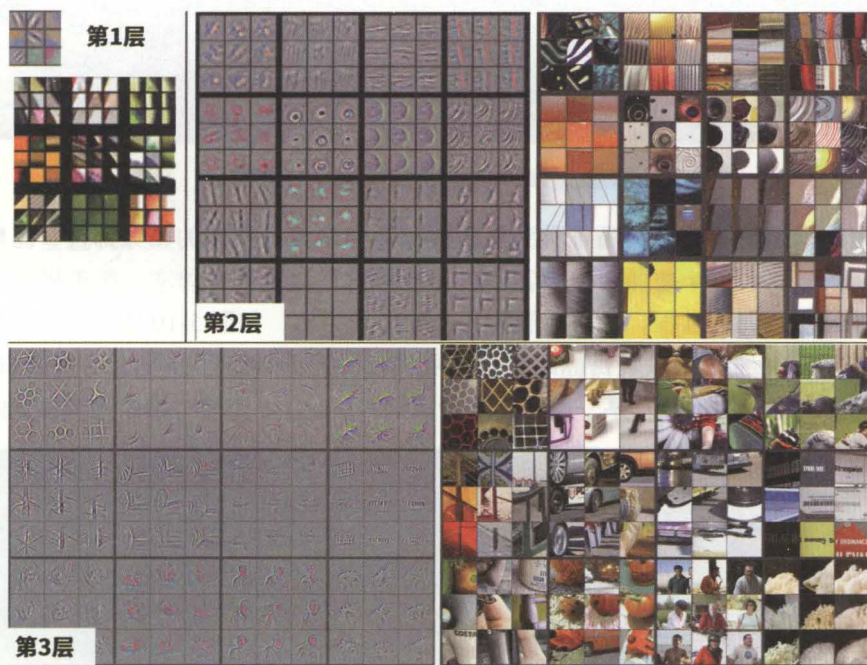


图 5-8 神经元的识别目标：第 1~3 层

可见，第 1 层的神经元可识别各种角度的边缘以及某些颜色，符合此前看到的 conv1 的卷积核的情况。第 2 层可识别简单的形状和纹理（这些特征是由第 1 层的边缘和颜色组成）。第 3 层可识别物体的更大更复杂的局部（是由第 2 层的概念组成）。由此类推，逐层对应着越来越复杂的概念，最后在高层可看出物体的分类情况。

例如，第 4 层和第 5 层的识别目标如图 5-9 所示。

① 地址为 <https://arxiv.org/pdf/1311.2901.pdf>。



图 5-9 神经元的识别目标：第 4~5 层

在左边的识别目标图中，还可看到每个神经元会聚焦到图像的相关部位，忽略图像的背景。

再看神经元的训练过程。以第 3 层的几个神经元为例，如图 5-10 所示，从左到右是随着训练的进程，神经元的识别目标的变化。可见，一开始神经元并没有明确的识别目标，但随后会逐渐成型，明确针对某一类目标进行识别：



图 5-10 神经元的识别目标：随着训练的变化

由于神经元是随机初始化，所以它们会有不同的演变方向，就像下山时朝着不同的路线前进，最终会分别识别不同类型的目标。

4) 可通过遮挡图像，发现哪些区域对于类别识别最为关键。具体方法是每次将图像的

一部分清零（在图 5-11 中显示为灰色），观察是否会影响正确的识别结果。影响越大，就说明挡住的区域越为重要。可将结果画成热力图。

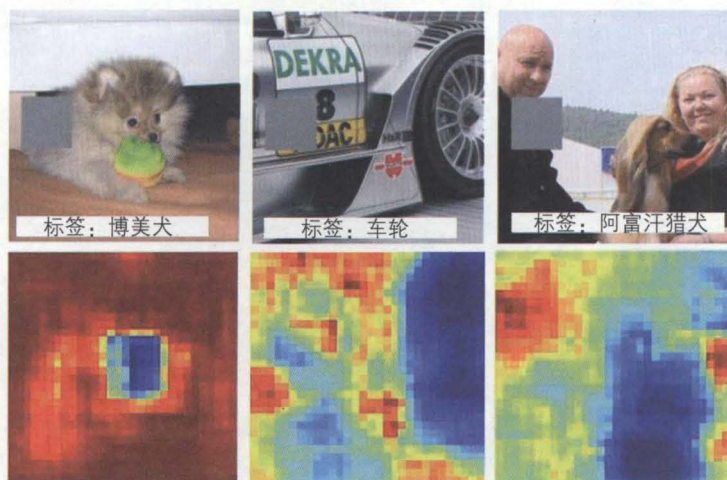


图 5-11 通过遮挡图像，发现关键区域

图中的蓝色代表最关键的区域，红色代表最无关紧要的区域。可见，博美犬的面部对于判断品种很重要，车轮对于判断车轮很重要，阿富汗猎犬的整个身体对于判断品种都很重要。

5) 可采用 2017 年 6 月发布的 SmoothGrad 技术 (<https://pair-code.github.io/saliency/>), 找到更细致的判断依据，见图 5-12。



a) 输入源图像

b) 网络判断为足球的原因所在

c) 正确答案马尔济斯犬的原因所在

图 5-12 通过 SmoothGrad，发现关键区域

可见，网络判断为足球是根据足球表面的六边形纹理，而正确答案马尔济斯犬来自于小狗的纹理。

6) 注意到 AlexNet 的 fc2 层的输出在 ReLU 后是 4096 个数字，可认为它们代表了图像的 4096 维的语义编码。通过使用 t-SNE 聚类，可以将这 4096 维降到 2 维，即，为每个图像找到一个 2 维坐标，使得坐标相近的图像对应于编码相近的图像。效果如图 5-13 所示。



图 5-13 t-SNE 聚类

可见，车辆、植物、动物等都会自动聚集在一起，且可观察到语义的平滑过渡。

7) 在 <http://people.csail.mit.edu/torralba/research/drawCNN/drawNet.html> 还提供了清晰的 AlexNet 网络运作示例，如图 5-14 所示。

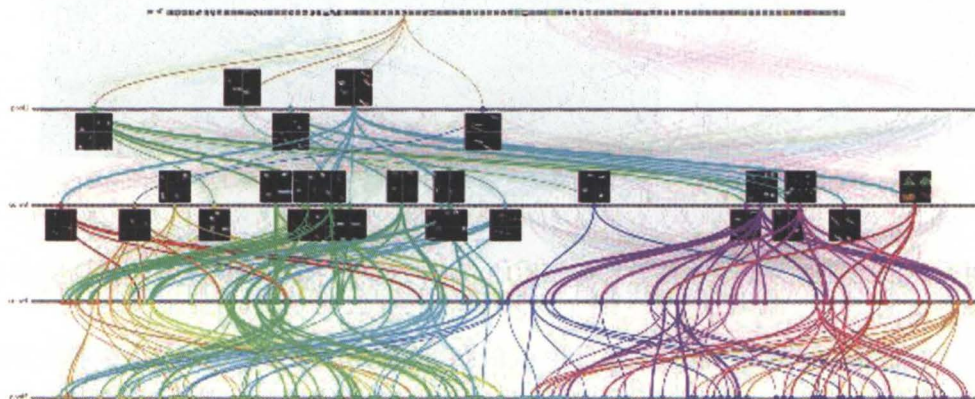


图 5-14 AlexNet 的运作

选定其中的某个卷积神经元，可直接看到它与其他卷积神经元之间的连接权重的可视化，以及最能激活这些卷积神经元的样本图像区域。

8) 通过 GAN 架构[⊖]，可更清晰和全面地展现某个神经元所学会判别的事物。如图 5-15 所示，某个神经元看上去是识别人脸，但经过分析，它实际也会对建筑的门窗，以及绿色产生反应。



图 5-15 通过 GAN 发现神经元的识别目标

图 5-15 是通过 GAN 技术生成的能激活这个神经元的图像。

5.3 迁移学习：精调、预训练等

随着深度学习的发展和网络架构的成熟，迁移学习的重要性在不断提高。我们先看迁移学习的思想来源。

在此前的训练中，我们每次都是从头训练一个随机初始化的网络，这可能需要很长时间，例如 ImageNet 的训练常常需要几个星期。如果训练的硬件设备不足，更是如此。

不过仔细想想，这并非最佳做法。不妨以人脑为例。如果把人脑认为是一个巨型神经网络，那么它每时每刻都在训练之中，我们并不会每次在学习一个新任务前把大脑清零重新初始化，而是恰好相反，正所谓触类旁通，不同领域之间的知识可以互相补充，互为促进。

这正是迁移学习的愿望，它希望能将一个问题上的训练所得，用于改进在另一个问题上的效果。目前迁移学习在深度学习中的常见做法是，先下载其他研究人员在大数据集（如 ImageNet）上训练过的网络模型，然后可做两件事情：

- 用我们的数据继续训练，这称为精调（fine-tuning）。
 - 这相当于让网络从一个良好的初始点开始学习，往往可以大大节省训练时间，特别是在两边的数据集相似的情况下。例如，将一个在 ImageNet 上训练好的分类模型，用于训练分类其他物体。
 - 精调的有效性，来自于深度网络中提取出的特征的普适性。例如网络第 1 层的滤

⊖ 地址为 <https://arxiv.org/pdf/1612.00005.pdf>。

波器，往往对于图像的边缘特征的提取，这对于几乎所有图像问题都很重要。

- 精调的另一优势是，往往可降低过拟合，因为网络已经学习过了大量的数据。于是我们可用更少的额外数据，训练出期望的模型。
- 可见，精调可以同时改善算力不足和数据不足。
- 用于特征提取，达成其他目的。
 - 由于深度网络可从数据中提取出优秀的特征，我们可将其用于其他的机器学习方法，如线性模型或 SVM 模型，如《CNN Features off-the-shelf: an Astounding Baseline for Recognition》[⊖]一文所显示的。
 - 一个典型例子是，移除 AlexNet 的最后一层，使用倒数第二层提取出的 4096 维编码。在此基础上可以做不少事情，例如前文曾说过，可用它判断图像在语义上的相似程度。
 - 后文也会有个例子：运用在 ImageNet 上训练好的 VGG 网络，将图像的风格做转移。

那么，在哪里可下载训练过的模型？可在 Google 搜索 Model Zoo（模型大全），或在 GitHub 上搜索网络的架构名称以及 checkpoint（人们往往用这个词代表训练过的模型数据）。

MXNet 的 Model Zoo 地址为 https://mxnet.incubator.apache.org/model_zoo/index.html，其中有在多个数据集上训练过的模型，如表 5-7 所示。

表 5-7 MXNet 的 Model Zoo

Model Definition	Dataset	Model Weights	Research Basis	Contributors
CaffeNet	ImageNet	Param File	Knzhersky,2012	@jispisak
Network in Network(NiN)	ImageNet	Param File	Lin et al.,2014	@jispisak
SquocozoNot v1.1	ImageNet	Param File	Landoia et al.,2016	@jispisak
VGG16	ImageNet	Param File	Simonyan et al.,2015	@jispisak
VGG19	ImageNet	Param File	Simonyan et al.,2015	@jispisak
Inception v3 wrBatchNorm	ImageNet	Param File	Szegedy et al.,2015	@jispisak
ResirtualNet152	ImageNet	Param File	HE et al.,2015	@jispisak
ResNext101-64x4d	ImageNet	Param File	Xie et al.,2016	@Jemyzcn
Fast-RCNN	PASCAL VOC	[Param File]	Cirshics,2015	
Faster-RCNN	PASCAL VOL	[Param File]	Ren et al.,2016	
Singk Shot Octnchon(SSD)	PASCAL VOL	[Param File]	Liu et al.,2016	
LocahonNet	MutbmodiaCommons	Param File	Weyand et al.,2016	@jychoiB4@kovin47

其实，即使训练数据已经很充分，也可先在其他数据集上训练，以改善过拟合，同时提高性能，这称为预训练（pre-training）。正如《Revisiting Unreasonable Effectiveness of

⊖ 地址为 <https://arxiv.org/abs/1403.6382>。

Data in Deep Learning Era》[⊖]所显示，采用3亿张训练图像作预训练后，各种网络模型的效果都会得到明显提升。而且看上去3亿张训练图像仍没有触及深度神经网络的极限，我们依旧会发现训练图像越多效果越好。

预训练的技巧：

- 如果额外数据很少，可固定网络的前几层，以防过拟合，因为网络的前几层对应的是更底层、更通用的特征。所以可只训练最后的一层或几层。
- 可使用相对更低的学习速率，因为网络的初始值已经不错。

与迁移学习相关的话题是一次学习和零次学习（one/zero-shot learning），许多研究人员认为这是使AI接近人类智能的重要待解问题。

一次学习，就是只需一个训练样本。例如，当小孩第一次看到猫，大人告诉它是猫，小孩一次就能掌握这个概念，以后看到猫时能认出来。

而零次学习的难度更大，就像在读到“猫有四条腿和尖耳朵”后，就能在未来看到一张猫的图像时推测出这可能是猫。严格说来，零次学习也需要额外的信息输入。

人类拥有较强的一次学习和零次学习能力。而目前的深度神经网络却不擅长这一点，它可能需要看过几千张甚至更多的图像才能大致建立图像中的概念。迁移学习也许是解决这个问题关键之一。

最后一个有趣的话题是教师-学生网络。我们可先训练一个很宽（即每层的神经元数很多）的网络，称为教师网络。这种网络容易训练，但运行速度会较慢。

然后训练一个更窄更深的网络，称为学生网络，这种网络更难训练。但我们可让学生网络同时预测目标输出和教师网络的中间层的输出，这相当于引入了某种正则化，也相当于将教师网络学到的中间表示迁移到学生网络中。经研究人员的实验，这往往可让训练变得更为快速和准确。

5.4 架构技巧：基本技巧

5.4.1 感受野与缩小卷积核

在前文提到，现代卷积网络中往往只会使用 3×3 或更小的卷积核，在此我们进一步讨论。

先看一个概念：感受野（receptive field）。经过1次 $k \times k$ 卷积后，输入图像的 $k \times k$ 像素会变成输出图像的 1×1 像素，所以输出图像的每1个像素与输入图像的 $k \times k$ 像素有关。如果再做1次 $k \times k$ 卷积，那么输出图像的每1个像素又和刚才得到的图像的 $k \times k$ 像素有关，于是就和最早的图像的 $(k \times 2 - 1) \times (k \times 2 - 1)$ 像素有关。

⊖ 地址为 <https://arxiv.org/abs/1707.02968>。

如图 5-16 所示，经过 2 层 3×3 卷积后，顶层图像的每 1 个像素将与中间层图像的 3×3 像素有关，与底层图像的 5×5 像素有关。

在多层卷积网络中，随着不断的卷积操作，输出的图像的每 1 个像素，会与输入图像的越来越多的像素有关。那么，如果输出图像的每 1 个像素和输入图像的 $n \times n$ 的像素有关，就称为输出图像的感受野是 $n \times n$ 。

图 5-16 还说明，可以用 2 层 3×3 卷积，模拟 5×5 卷积。严格来说，这样做只能模拟比较对称的 5×5 ，因为 5×5 的参数量是 25，比 3×3 的参数量 9 的 2 倍还要多（在此忽略偏置）。

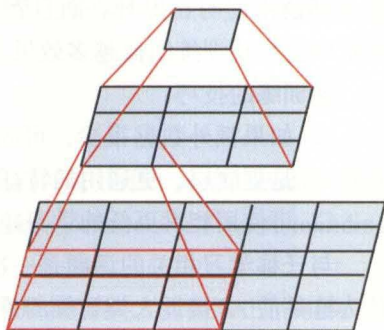
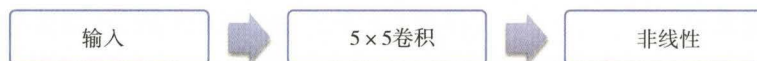


图 5-16 3×3 卷积的叠加

不过，这也意味着，如果用 2 层 3×3 可以达到类似 1 层 5×5 的效果，那么就可以节省参数量，参数节省的程度是 $(25-18)/25=28\%$ 。

而且，我们可在每 1 层 3×3 后都加入非线性，这就不仅仅是模拟 5×5 ，还可实现更多的非线性，如图 5-17 所示。

原方法：



新方法：



图 5-17 拆分 5×5 卷积

进一步继续这个思路，就是著名的 Inception 架构，会在后文介绍。根据 Google 的实验，将卷积核拆分并加入更多的非线性，有可能进一步提升网络的性能。

例如，甚至可将 3×3 进一步拆分为 1×3 和 3×1 两层，如图 5-18 所示。

在传统的深度卷积网络架构中，如 AlexNet，会使用较大的卷积核（如 11×11 ），因为当时研究人员的观念是，卷积核越大，感受野越大，看到的图像信息更多，有可能获得更好的特征。

而在现代架构中，会改用多层 3×3 ，感受野同样大，可以节省参数，减少过拟合。而且由于网络更深，加入了更多非线性，效果往往更好。事实上，在深度学习中，模型往往是越深越好。

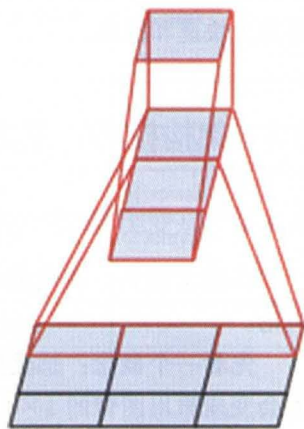


图 5-18 拆分 3×3 卷积

5.4.2 使用 1×1 卷积核

最小的卷积核是 1×1 卷积核。表面看来， 1×1 卷积核似乎不是真正的滤波器，它只是

将图像的每个点都乘以 1 个权重, 再加上 1 个偏置。

但这就与普通 MLP 网络中的神经元很像, 如图 5-19 所示。

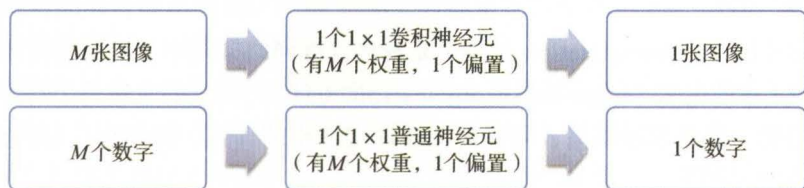


图 5-19 1×1 卷积与 MLP 网络的对比

因此, 通过 N 个 1×1 卷积神经元, 只需少量参数, 就能将 M 张图像变为 N 张图像。这通常有 4 种应用场景:

1) 如需将图像分为 N 类, 可在最后用 1×1 卷积层将 M 张图像转换为 N 张图像, 再通过全局池化变为 N 个数字, 送入 SoftMax 作为输出。

2) 可用 1×1 卷积层作为瓶颈层 (bottleneck layer)。举例说明。假设输入通道是 256 个, 要求经过 3×3 卷积, 最后输出通道也是 256 个, 那么有 2 种实现方法:

□ 第 1 种: 直接送入 3×3 卷积层, 如图 5-20 所示。

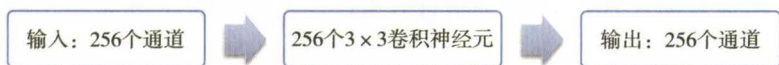


图 5-20 未使用瓶颈层的网络

○ 忽略偏置, 参数量是 $256 \times 3 \times 3 \times 256 = 589824$ 。

□ 第 2 种: 先送入 1×1 卷积层变为 64 个通道, 再送入 3×3 卷积层维持为 64 个通道, 再送入 1×1 卷积层变为 256 个通道, 如图 5-21 所示。

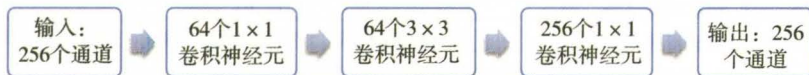


图 5-21 使用瓶颈层的网络

○ 忽略偏置, 参数量是 $256 \times 1 \times 1 \times 64 + 64 \times 3 \times 3 \times 64 + 64 \times 1 \times 1 \times 256 = 69632$, 仅为此前的 11.8%。

○ 而且, 由于网络更深, 可在各层之间插入更多的非线性, 网络的表达能力仍然很强。

可见, 我们可灵活地在通道数很多的卷积层之间, 插入通道数更少的 1×1 卷积层, 作为瓶颈层。

3) 只要我们希望改变通道数或图像尺寸, 简单的方法就是用 1×1 卷积层。例如在后文的残差网络中就会用到这一点。

4) 通过连续使用多个 1×1 卷积层, 可在图像的每个点上实现 1 个小型的 MLP 网络。

这是《Network In Network》[⊖]的思想。

5.4.3 批规范化

批规范化 (Batch Normalization, BN) 是现代深度网络架构中最常用的技巧之一, 来自 Google 在 2015 年的论文 (<https://arxiv.org/abs/1502.03167>)。它的原理是让网络中间数据的分布尽量规范化, 就像校准准星, 可显著加速深度网络的训练, 并拥有一定的正则化能力, 可改善过拟合。

我们通常会将 BN 层放在非线性激活层的前面, 形成如图 5-22 所示的架构。

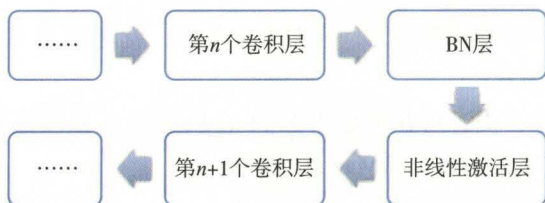


图 5-22 将 BN 层加入网络

如前文所述, 如果用 sigmoid 作为非线性激活, 容易出现梯度消失, 因此不利于训练深度神经网络。但通过加入 BN, 可明显增进效果。以某个采用 sigmoid 激活的 MNIST 网络为例, 效果如图 5-23 所示。

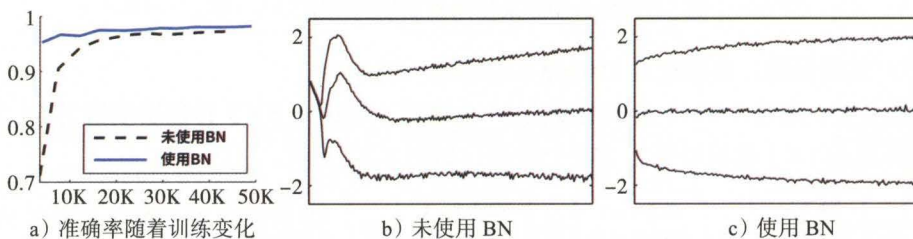


图 5-23 BN 层的效果

由图 a 可见, 使用 BN 后, 训练速度明显更快, 准确率更高。

图 b 和 c 是网络中某条路径的数据分布随着训练的变化情况, 3 条曲线分别代表数据分布的 15% 位置、50% 位置和 85% 位置。可见, 使用 BN 后分布更稳定、对称, 且均值更接近 0, 这有利于神经元的训练和改善过拟合。

下面看 BN 的原理。我们举例说明深度神经网络训练中的一个难点, 以及 BN 在此的作用。

考虑一个最简化的 3 层“深度神经网络”, 每层只有一个神经元, 只有一个输入, 且没有偏置, 没有非线性激活。令输入为 x , 输出为 $y = x \cdot w_1 \cdot w_2 \cdot w_3$ 。那么考虑 2 组参数:

[⊖] 地址为 <https://arxiv.org/abs/1312.4400>。

□ $w_1=1, w_2=1, w_3=1$, 则 $y=x$ 。

□ $w_1=100, w_2=0.0001, w_3=100$, 则也有 $y=x$ 。

这2组参数的最终输出相同,但在训练中的稳定性有很大区别。例如,对于第2组参数,若将参数做少许变动,就会对输出造成很大的影响。

我们可在深度网络中加入额外的保护措施,避免出现第2组参数的情况,但这往往会带来大量额外计算,使训练速度减慢,最终得不偿失。而BN仅需少量计算即可改善这一点。

那么BN是如何实现这一点的?注意到第2组参数的一个特点:中间层输入的数值容易很大或很小,这都不利于神经元的学习,会造成极端的参数。

于是,简单的想法是希望让神经元的每个输入通道(如果通道是图像,则取图像所有点的平均值)尽量满足均值为0,方差为1的分布。但这种分布并不一定适合后续的非线性激活函数,所以BN为每个通道再引入2个可训练的额外参数 β 和 γ ,初始值为 $\beta=0, \gamma=1$,然后BN让每个通道尽量满足均值为 β ,方差为 γ 的分布,且 β 和 γ 对于每个通道可以不一样。因此,BN层会给每个通道增加2个参数。

具体实现方法:

□ 在训练网络时,计算每个通道在每一个batch中的均值和方差,然后可做线性变换将其移动到所需的分布中。

□ 参数 β 和 γ 的训练方法可通过求偏导计算得到。

□ 在运行网络时,可采用训练时收集的均值 m 和方差 v ,完成同样工作。具体公式如下,其中 x 为BN层输入, y 为BN层输出, ε 通常取0.001或0.00001,以保证分母不为0。

$$y = \gamma \cdot \frac{x - m}{\sqrt{v + \varepsilon}} + \beta$$

在MXNet中定义BN层是通过`mx.sym.BatchNorm`,它的`fix_gamma`参数代表是否将 r 固定为1,我们通常会将`fix_gamma`设为`False`。

在加入BN后,我们还可增大学习速率,因为梯度爆炸的几率得到了降低。

5.4.4 实例:回顾Fashion-MNIST问题

本节的网络仅有214 890个参数,配合数据增强,在Fashion-MNIST上可实现94.8%的准确率。代码如下:

```
#-*-coding:utf-8-*-

import numpy as np
import os
import gzip
import struct
import logging
import mxnet as mx
import matplotlib.pyplot as plt # 这是常用的Python绘图库
```



```

logging.getLogger().setLevel(logging.DEBUG)

def read_data(label_url, image_url): # 读入训练数据
    with gzip.open(label_url) as flbl: # 打开标签文件
        magic, num = struct.unpack(">II", flbl.read(8)) # 读入标签文件头
        label = np.fromstring(flbl.read(), dtype=np.int8) # 读入标签内容
    with gzip.open(image_url, 'rb') as fimg: # 打开图像文件
        magic, num, rows, cols = struct.unpack(">IIII", fimg.read(16)) # 读入图像
        # 文件头
        image = np.fromstring(fimg.read(), dtype=np.uint8) # 读入图像内容
        image = image.reshape(len(label), 1, rows, cols) # 设置为正确的数组格式
        image = image.astype(np.float32)/255.0 # 归一化到0到1区间
    return (label, image)

# 读入数据
(train_lbl, train_img) = read_data('train-labels-idx1-ubyte.gz', 'train-images-
idx3-ubyte.gz')
(val_lbl, val_img) = read_data('t10k-labels-idx1-ubyte.gz', 't10k-images-idx3-
ubyte.gz')

batch_size = 32 # 批大小

# 用于辅助定义网络，这是深度卷积网络中常用的"卷积-BN-非线性"模块
def CBA(src, suffix, num_filter, kernel, pad):
    conv = mx.sym.Convolution(data=src, name="conv"+suffix, kernel=
        (kernel, kernel), pad=(pad, pad), num_filter=num_filter)
    bn = mx.sym.BatchNorm(data=conv, name="bn"+suffix, fix_gamma=False)
    act = mx.sym.Activation(data=bn, name="act"+suffix, act_type="relu")
    return act

# 由于辅助定义网络，这里在每2个卷积层后加1个池化层
def LAYER(src, layer, num_filter, pad):
    conv1 = CBA(src, layer+"1", num_filter, 3, pad)
    conv2 = CBA(conv1, layer+"2", num_filter, 3, pad)
    pool = mx.sym.Pooling(data=conv2, name="pool"+layer, pool_type="max",
        kernel=(2,2), stride=(2,2))
    return pool

# 设置网络
data = mx.symbol.Variable('data')
# 将3*28*28变换为32*14*14
net = LAYER(data, "1", 32, 1)
# 将32*14*14变换为64*7*7
net = LAYER(net, "2", 64, 1)
# 将64*7*7变换为64*3*3
net = LAYER(net, "3", 64, 1)
# 将64*3*3变换为128*1*1
net = CBA(net, "4", 128, 3, 0)
# 将128*1*1变换为10*1*1
net = mx.sym.Convolution(data=net, name="final", kernel=(1,1), num_filter=10)
# 将10*1*1变换为10
net = mx.sym.Flatten(data=net, name="flatten")
net = mx.sym.SoftmaxOutput(data=net, name='softmax')

# 输出参数情况供参考

```

```

shape = {"data" : (batch_size, 1, 28, 28)}
mx.viz.print_summary(symbol=net, shape=shape)

# 由于训练数据多，这里采用了GPU，若读者没有GPU，可修改为CPU
module = mx.mod.Module(symbol=net, context=mx.gpu(0))

# 迭代器：测试数据
val_iter = mx.io.NDArrayIter(val_img, val_lbl, batch_size)

# 手动循环40个epoch
for epoch in range(40):
    # 生成增强的图像，最佳方法是在另一进程执行，这里只是演示
    # 首先复制一份原始图像
    aug_img = train_img.copy()
    # 修改其中的每幅图像
    for i in range(aug_img.shape[0]):
        # 有50%概率做左右翻转
        if np.random.random() < 0.5:
            # aug_img[i][0]为第i号样本的0号通道，灰度图像只有0号通道
            # fliplr()用于左右翻转
            aug_img[i][0] = np.fliplr(aug_img[i][0])

        # 左右移动最多2个像素，注意randint(a,b)的范围为a到b-1
        amt = np.random.randint(0, 3)
        if amt > 0: # 如果需要移动...
            if np.random.random() < 0.5: # 左移动还是右移动?
                # pad()用于加上外衬，因移动后减少的区域需补零
                # 然后用[:]取所要的部分
                aug_img[i][0] = np.pad(aug_img[i][0], ((0,0),(amt,0)),
                                         mode='constant')[:, :-amt]
            else:
                aug_img[i][0] = np.pad(aug_img[i][0], ((0,0),(0,amt)),
                                         mode='constant')[:, amt:]

        # 上下移动最多2个像素
        amt = np.random.randint(0, 3)
        if amt > 0:
            if np.random.random() < 0.5:
                aug_img[i][0] = np.pad(aug_img[i][0], ((amt,0),(0,0)),
                                         mode='constant')[:-amt, :]
            else:
                aug_img[i][0] = np.pad(aug_img[i][0], ((0,amt),(0,0)),
                                         mode='constant')[amt:, :]

        # 随机清零最大5*5的区域
        x_size = np.random.randint(1, 6)
        y_size = np.random.randint(1, 6)
        x_begin = np.random.randint(0, 28-x_size+1)
        y_begin = np.random.randint(0, 28-y_size+1)
        aug_img[i][0][x_begin:x_begin+x_size, y_begin:y_begin+y_size] = 0

    # 每个epoch重设训练数据
    train_iter = mx.io.NDArrayIter(aug_img, train_lbl, batch_size, shuffle=True)
    # 每个epoch降低学习速率
    lr = 0.06 * pow(0.95, epoch)

```

```

# 输出当前epoch信息
print("epoch " + str(epoch) + ", learning rate = " + str(lr))
# 训练
module.fit(
    train_iter,
    eval_data=val_iter,
    optimizer = 'sgd',
    optimizer_params = {'learning_rate' : lr},
    num_epoch = 1, # 每次训练1个epoch
    batch_end_callback = mx.callback.Speedometer(batch_size, 60000/batch_size)
)

```

训练输出如下：

```

epoch 0, learning rate = 0.06
INFO:root:Epoch[0] Train-accuracy=0.813783
INFO:root:Epoch[0] Time cost=17.857
INFO:root:Epoch[0] Validation-accuracy=0.874301
epoch 1, learning rate = 0.057
INFO:root:Epoch[0] Train-accuracy=0.874400
INFO:root:Epoch[0] Time cost=17.837
INFO:root:Epoch[0] Validation-accuracy=0.906450
.....
epoch 39, learning rate = 0.00811655725674
INFO:root:Epoch[0] Train-accuracy=0.952000
INFO:root:Epoch[0] Time cost=17.824
INFO:root:Epoch[0] Validation-accuracy=0.948383

```

由于这里使用了有效的数据增强方法，过拟合得到了很好的控制，在训练集的准确率为 95.2%，在测试集的准确率也达到了 94.8%。

如果扩大网络的尺寸，例如将每层的卷积单元设置为 32-64-128-256，可得到更高的准确率，不过计算量和参数量也会明显增加。在实际中，我们往往需要找到一个在速度与准确率之间取得平衡的模型，这也与模型的具体应用场景有关。

5.4.5 实例：训练 CIFAR-10 模型

由于 ImageNet 的数据量高达 100G 以上，训练耗时很长，对训练硬件的要求很高。本节我们看另一个难度适中的经典问题：著名的 CIFAR-10 数据集，其主页是：<http://www.cs.toronto.edu/~kriz/cifar.html>。

CIFAR-10 共包括 60000 张 32×32 的彩色图像，其中有 50000 张训练图像，10000 张测试图像，目标是将图像分为如图 5-24 所示的 10 类。

这些类别之间是互斥的。例如，汽车类别包括轿车、SUV 等；卡车类别只包括大卡车；两者都不包括皮卡。

经实际测试，其中最难分辨的是猫和狗（由于图像很小，有时确实不容易分辨），飞机和船（注意天空和水面都是蓝色），汽车和卡车（在概念上就很相近）。

虽然我们可直接下载二进制格式的 CIFAR-10 数据集，但笔者希望在此让读者看到从图像文件到训练的全过程，这样读者可用类似的方法训练自己的数据集。



图 5-24 CIFAR-10 中的图像

首先，下载 CIFAR-10 的 PNG 图像数据：<https://pjreddie.com/projects/cifar-10-dataset-mirror/>，解压后会得到 60000 张 PNG（分别位于 test 和 train 目录），以及 labels.txt。

下面将图像打包为 MXNet 的 RecordIO 格式，可参阅前文的说明。首先，需将图像文件按分类标签移动到相应的子目录中，可通过下列代码实现：

```
#-*-coding:utf-8-*-
```

```
import os
# 读入标签列表
file = open("labels.txt", "r")
labels = file.readlines()
for i in range(len(labels)):
    labels[i] = labels[i].strip()
print(labels)
# 需移动的图像所在目录，记得运行后修改为test并再次运行，以移动测试图像
target = './train/'
# 新建10个子目录用于存放相应类别的图像
for i in range(10):
    directory = target + str(i)
    if not os.path.exists(directory):
        os.makedirs(directory)
# 开始移动图像
```

```

for filename in os.listdir(target):
    if not filename.endswith('.png'):
        continue
    print(filename)
    tmp = filename.split('_')
    # 移除 ".png" 后缀后找到图像的类别号
    index = labels.index(tmp[1][:-4])
    # 移动图像
    os.rename(target + filename, target + str(index) + '/' + filename)

```

将 train 和 test 都移动后，目录结构如图 5-25 所示。



图 5-25 待打包的目录结构

假设根目录是 C:/cifar，那么如前文所述，可如下打包训练图像：

```

python im2rec.py --list=1 --recursive=1 C:/cifar/train C:/cifar/train
python im2rec.py --pass-through=1 C:/cifar C:/cifar/train

```

将得到的 train.idx、train.lst、train.rec 移动到程序目录下（以防重新打包），再继续打包测试图像：

```

python im2rec.py --list=1 --recursive=1 C:/cifar/test C:/cifar/test
python im2rec.py --pass-through=1 C:/cifar C:/cifar/test

```

再将得到的 test.idx、test.lst、test.rec 移动到程序目录下。

下面看代码。在此我们将图像裁剪成 28×28 ，网络架构与《Striving for Simplicity: The All Convolutional Net》^①一文类似，采用纯卷积架构，在层间用带步长的卷积代替池化层进行缩小，并在最后用全局池化层缩小：

① 地址为 <https://arxiv.org/abs/1412.6806>。

```

#-*-coding:utf-8-*-

import numpy as np
import logging
import mxnet as mx

logging.getLogger().setLevel(logging.DEBUG)

batch_size = 50 # 批大小

# 用于辅助定义网络，这是深度卷积网络中很常用的"Conv-BN-Act"模块
def CBA(src, suffix, num_filter, kernel, pad, stride=(1,1)):
    conv = mx.sym.Convolution(src, name="conv"+suffix, kernel=(kernel,kernel),
        pad=(pad,pad), num_filter=num_filter, stride=stride)
    bn = mx.sym.BatchNorm(conv, name="bn"+suffix, fix_gamma=False)
    act = mx.sym.Activation(bn, name="act"+suffix, act_type="relu")
    return act

# 全卷积架构，由带步长的卷积实现缩小
net = mx.symbol.Variable('data')
# 将3*28*28变换为96*28*28
net = CBA(net, "1", 96, 3, 1)
# 将96*28*28变换为96*28*28
net = CBA(net, "2", 96, 3, 1)
# 将96*28*28变换为96*14*14
net = CBA(net, "3", 96, 3, 1, (2,2))
# 将96*14*14变换为192*14*14
net = CBA(net, "4", 192, 3, 1)
# 将192*14*14变换为192*14*14
net = CBA(net, "5", 192, 3, 1)
# 将192*14*14变换为192*7*7
net = CBA(net, "6", 192, 3, 1, (2,2))
# 将192*7*7变换为192*5*5
net = CBA(net, "7", 192, 3, 0)
# 将192*5*5变换为192*5*5
net = CBA(net, "8", 192, 1, 0)
# 将192*5*5变换为10*5*5
net = CBA(net, "9", 10, 1, 0)
# 将10*5*5变换为10*1*1
net = mx.sym.Pooling(net, name="pool", global_pool=True, pool_type="avg",
    kernel=(1,1))
# 将10*1*1变换为10
net = mx.sym.Flatten(net, name="flatten")
net = mx.sym.SoftmaxOutput(net, name='softmax')

# 输出参数情况供参考
shape = {"data" : (batch_size, 3, 28, 28)}
mx.viz.print_summary(symbol=net, shape=shape)

# 由于训练数据多，这里采用了GPU，若读者没有GPU，可修改为CPU
module = mx.mod.Module(symbol=net, context=mx.gpu(0))

# 迭代器，训练数据：
train_iter = mx.io.ImageRecordIter(
    path_imgrec = "train.rec",

```



```

        data_shape = (3,28,28), # 图像通道和尺寸
        batch_size = batch_size,
        shuffle = True, # 开启随机次序
        rand_crop = True, # 开启随机裁剪
        rand_mirror = True, # 开启随机镜像
        random_h = 10, # 随机色相
        random_s = 20, # 随机饱和度
        random_l = 25, # 随机亮度
        max_random_scale = 1.20, # 随机放大
        min_random_scale = 0.88, # 随机缩小
        max_rotate_angle = 20, # 随机旋转
        max_aspect_ratio = 0.15, # 随机长宽比例
        max_shear_ratio = 0.10, # 随机倾斜比例
        fill_value = 0, # 四周填充黑色
    )
    # 测试数据, 关闭数据增强:
    val_iter = mx.io.ImageRecordIter(
        path_imgrec = "test.rec",
        data_shape = (3,28,28),
        batch_size = batch_size,
        shuffle = False,
        rand_crop = False,
        rand_mirror = False,
    )
    # 训练
    module.fit(
        train_iter,
        eval_data=val_iter,
        initializer = mx.init.MSRAPrelu(slope=0.0), # 采用MSRAPrelu初始化
        optimizer = 'sgd',
        # 采用0.5的初始学习速率, 并在每50000个样本后将学习速率缩减为之前的0.98倍
        optimizer_params = {'learning_rate' : 0.5, 'lr_scheduler' : mx.lr_scheduler.
            FactorScheduler(step=50000/batch_size, factor=0.98)},
        num_epoch = 200,
        batch_end_callback = mx.callback.Speedometer(batch_size, 50000/batch_size)
    )

```

训练输出如下:

```

INFO:root:Epoch[0] Train-accuracy=0.439880
INFO:root:Epoch[0] Time cost=23.359
INFO:root:Epoch[0] Validation-accuracy=0.533200
INFO:root:Update[1001]: Change learning rate to 4.90000e-01
INFO:root:Epoch[1] Train-accuracy=0.616120
INFO:root:Epoch[1] Time cost=23.641
INFO:root:Epoch[1] Validation-accuracy=0.681500
INFO:root:Update[2001]: Change learning rate to 4.80200e-01
.....
INFO:root:Update[197001]: Change learning rate to 9.34344e-03
INFO:root:Epoch[197] Train-accuracy=0.975100
INFO:root:Epoch[197] Time cost=23.174
INFO:root:Epoch[197] Validation-accuracy=0.921100
INFO:root:Update[198001]: Change learning rate to 9.15657e-03
INFO:root:Epoch[198] Train-accuracy=0.975820
INFO:root:Epoch[198] Time cost=23.151
INFO:root:Epoch[198] Validation-accuracy=0.925200

```

```
INFO:root:Update[199001]: Change learning rate to 8.97344e-03
INFO:root:Epoch[199] Train-accuracy=0.976740
INFO:root:Epoch[199] Time cost=23.151
INFO:root:Epoch[199] Validation-accuracy=0.923000
```

此网络使用了 1 372 254 个参数，最终测试准确率达到 92.4% 左右，对于这个参数量是不错的成绩。

由于这里的训练准确率很高，说明主要障碍是过拟合。读者可实验更强的数据增强，更多的正则化，以及进一步调参。我们在后文还会训练更先进的架构。

5.5 架构技巧：残差网络与通道组合

5.5.1 残差网络：ResNet 的思想

深度学习的一大原则是，神经网络越深，效果往往越好。但极深的网络并不容易训练。从 2012 年到 2014 年，从 8 层的 AlexNet 到 19 层的 VGG，神经网络的层数增长曾遇到瓶颈，梯度消失和梯度爆炸曾是困扰研究人员的难题。

如图 5-26 所示，如只使用普通架构的深度卷积网络，56 层网络的错误率反而比 20 层网络的错误率更高。

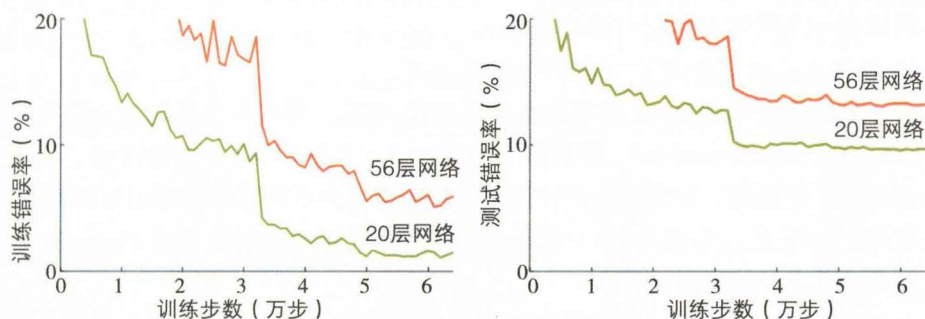


图 5-26 普通网络：更深不一定更好

一切在 2015 年改变。微软亚洲研究院 (MSRA) 公布的 152 层残差网络 (Residual Network, ResNet, 论文为《Deep Residual Learning for Image Recognition》^①), 在 ImageNet 2015 竞赛上以绝对优势取得冠军，这是深度网络架构的重要突破。

残差网络的秘诀是加入残差连接 (residual connection)，相当于建立信息高速公路，然后深度网络的错误率即可随着网络加深而不断降低，如图 5-27 所示。

① 地址为 <https://arxiv.org/abs/1512.03385>。

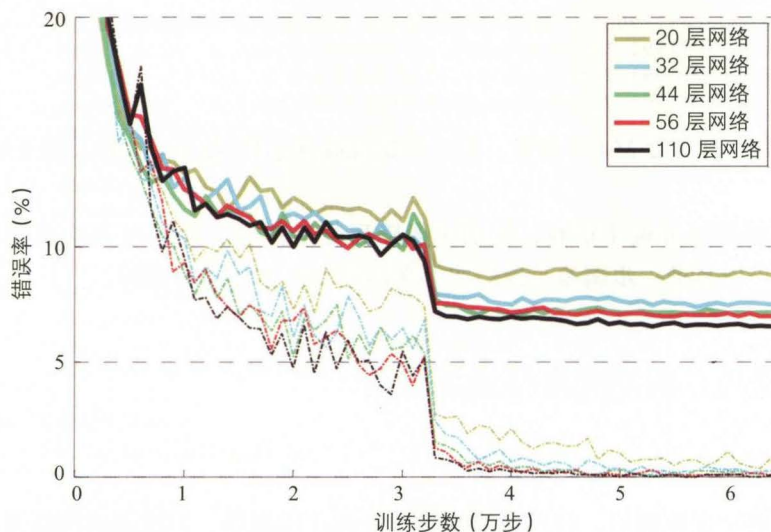


图 5-27 残差网络：越深越好

在 2016 年的后续改进中 (<https://arxiv.org/abs/1603.05027>), MSRA 更是成功训练了深达 1001 层的网络, 并实现了更低的错误率。

残差连接的思想, 其实很简单:

- 假设有一个网络 M , 它的输出是 X 。
- 给 M 加上一层变为 M' , 令新的输出是 $H(X)$ 。
- 有一种办法可以保证 M' 的性能至少不会比 M 差, 那就是令新加的一层为恒等变换 (identity transformation), 即直接令 $H(X)=X$ 。于是 M' 和 M 的输出就一模一样, 性能相同。这说明, 如果合理设计网络架构, 它至少不应随着网络的加深而变差。
- 更好的办法是, 令 $H(x)=x+F(x)$, 然后只要求新的一层去学习 $F(x)$, 这里的 $F(x)$ 就称为残差 (residue)。
 - 那么新的一层的学习难度就减轻了许多, 因为 “ $F(x)$ 恒等于 0” 已经是不错的结果, 至少不会令网络的性能降低。因此, 只要新的一层学会比 “恒等于 0” 更好的函数, 它就可以实现网络性能的提高, 换言之, M' 几乎一定会比 M 更好。
 - 注意, 这里的加法是点对点加法, 需要两边的通道数和图像尺寸都一致。若不一致, 可加上相应的 1×1 卷积层做调整。

于是这就实现了网络越深越好的梦想。残差连接已成为深度神经网络架构中的重要组成部分:

- 在引入残差连接后, 信息就直接拥有了跳过网络中某些层的选择权, 这有利于信息在网络中的流动。在后续研究中, 研究人员也观察到了这种现象。
- 在原始的神经网络中, 信息也拥有这种选择权。残差结构的意义是让这种选择变得更直接, 更容易被网络在训练中发现。

实际而言，网络的深度还有3个限制：第一，网络越深，运行速度越慢，对设备的内存要求也越高；第二，这种进步会越来越慢（在之前的图中也可看到这个现象）；第三，有可能会过拟合。因此，目前实际运用的网络仍然多数在一百层以内。

5.5.2 残差网络：架构细节

经实际测试，每次跳过2层网络可实现显著更好的性能，因为2层网络可提供更多的非线性，拟合更复杂的 $Y=F(x)$ 。因此，在2015年的ResNet原始论文中，实际做法如图5-28所示。

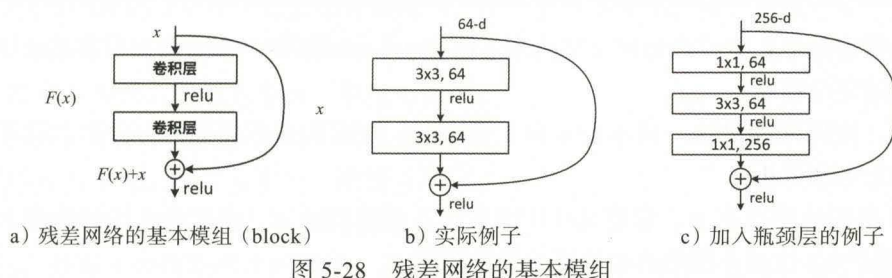


图 5-28 残差网络的基本模组

注意，在ResNet的原始论文中，将ReLU非线性激活放置在 $X+F(X)$ 之后，那么，即使 $F(X)=0$ ， X 仍然会经过1次ReLU。因此它并没有建立真正畅通的信息高速公路。当时研究人员也发现，如果用这种架构，则1202层的网络的性能不如110层的网络。

而在2016年的后续论文中（《Identity Mappings in Deep Residual Networks》[⊖]），研究人员测试了多种网络的组合方式，包括BN层和非线性层的位置调整，如图5-29所示。

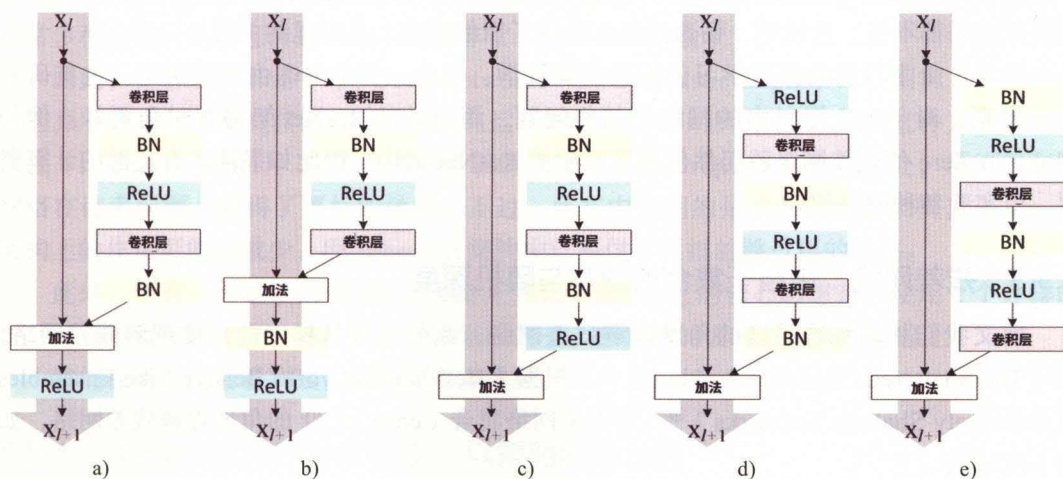


图 5-29 残差网络模组的调整

[⊖] <http://arxiv.org/abs/1603.05027>

上述几种架构，运用在 110 层和 164 层网络后，在 CIFAR-10 上的错误率如表 5-8 所示。

表 5-8 不同残差网络模组的性能

结构	在 110 层网络的错误率	在 164 层网络的错误率
a: 原始 ResNet	6.61	5.93
b	8.17	6.50
c	7.84	6.14
d	6.71	5.91
e: Pre-act ResNet	6.37	5.46

在错误率最低，效果最好的 e 架构中（称为 pre-act 架构），左通路上没有 ReLU 层，实现了信息的完全畅通。

不过，在同样满足这一要求的 c 和 d 架构中，性能就并不如意，为什么会这样？这需要仔细的思考和分析：

❑ c 架构的问题在于，它将 ReLU 层放在右通路的最后。由于 ReLU 的输出永远大于等于 0，这对于网络的表达能力是有害的。

❑ 而 d 和 e 的区别在于 BN 层的位置。如前所述，BN 层最适宜直接放在非线性激活层的前面。这里的 e 无疑更满足这一要求。

通过使用 e 架构，研究人员成功训练出了性能优秀的 1001 层网络，在多个测试数据集上实现了当时世界最佳的成绩。

研究人员还尝试了多种其他调整，例如在左通路加入 1×1 卷积（由于 1×1 卷积可包括恒等连接，理论上可能实现更强的表达能力）或 DropOut（理论上可能进一步降低过拟合），但实际效果都不佳。这说明，畅通无阻、直截了当的通路，是这里的关键。

注意，由于目前的深度学习仍然是实验科学，因此一切最终需由实验决定。也有研究人员发现，对于 100 层以内的网络和某些问题，用原始的 ResNet 架构有时会更好。例如 AlphaGo Zero 的残差网络就仍然使用了原始的 ResNet 架构。因此如果读者有足够的计算资源，可实验各种网络架构。

5.5.3 残差网络：来自于集合的理解与随机深度

前文我们从训练难易度的角度理解残差网络，此外也可从梯度的角度理解残差网络，见《The Shattered Gradients Problem》[⊖]。根据《Residual Networks Behave Like Ensembles of Relatively Shallow Networks》[⊖]，还可从网络集合（ensemble）的角度理解残差网络，如图 5-30 所示。

⊖ 地址为 <https://arxiv.org/abs/1702.08591>。

⊖ 地址为 <https://arxiv.org/abs/1605.06431>。

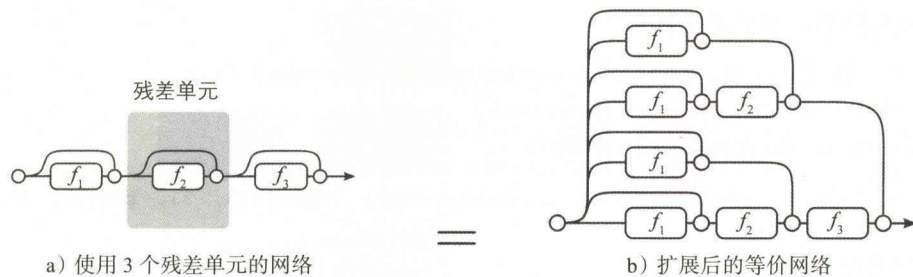


图 5-30 残差网络的等价变换

对于图 5-30a 的具有 3 个残差单元的网络，由于加入了残差连接，数据在从左到右的过程中有多种路径，因此实际等价于如图 5-30b 所示的网络。可举例说明其中的路径：

- 跳过 f_1 单元，跳过 f_2 单元，跳过 f_3 单元。
- 经过 f_1 单元，跳过 f_2 单元，跳过 f_3 单元。
- 经过 f_1 单元，经过 f_2 单元，跳过 f_3 单元。
- 以此类推，总共有 $2^3=8$ 种路径。

可见，具有 n 个残差单元的网络，就相当于是 2^n 个浅层网络的集合。由于网络的集合往往会效果更好，过拟合更少，可见残差结构也有改善过拟合的功效。

为了进一步证实这一点，研究人员实验了直接删除深度残差网络中的部分层，发现网络对此有很好的抵抗能力，性能下降较小。这说明，残差网络的架构的确更可靠。

同时，研究人员还分析了深度残差网络中梯度的传导情况，发现对于一个具有 54 个残差单元的网络，每个 batch 样本的梯度往往只会经过其中的 5 到 17 个单元。结合其他证据，研究人员最终认为，也许深度残差网络的威力不是因为它可以非常深，而是在于它提供了非常多种路径可供信息传递，相当于是非常多个网络的集合。

因此，在后续的研究中（《Deep Networks with Stochastic Depth》^①），研究人员在训练时选择有随机的几率直接跳过部分层（在运行时仍然会使用所有层），并称这种方法为随机深度（stochastic depth）。这可提高网络训练速度，且与 DropOut 有类似之处，最终得到的网络在许多问题上获得了更好的效果。不过，此文中采用的是原始的残差架构，在 pre-act 架构上的作用可能会减少，因为 pre-act 架构中已提供了无损的跳过层的通路。

此外，还有一种最新的 Shake-Shake 正则化方法^②，它的核心思想是，使用 2 个并排的层，并且不直接跳过层，而是随机加入衰减因子。感兴趣的读者可研究和实验。

我们在后文还会看到残差网络的更多进展。

5.5.4 残差网络：MXNet 实现，以策略网络为例

让我们举例说明残差网络在 MXNet 中的实现。本书将训练的围棋策略网络就是一个

① 地址为 <https://arxiv.org/abs/1603.09382>。

② 地址为 <https://arxiv.org/pdf/1705.07485.pdf>。

pre-act 残差网络，它的定义如下：

```
# 每层的卷积神经元个数，AlphaGo Zero使用256个，我们为加快训练使用128个
n_filter = 128

net = mx.symbol.Variable('data')
# 预处理
net = mx.sym.Convolution(net, name='convPRE', kernel=(3, 3), pad=(1, 1), num_filter = n_filter)

# 残差结构
for i in range(0, 6):
    identity = net # 保存之前的输出
    net = mx.sym.BatchNorm(net, name='bnA'+str(i), fix_gamma=False)
    net = mx.sym.Activation(net, name='actA'+str(i), act_type='relu')
    net = mx.sym.Convolution(net, name='convA'+str(i), kernel=(3, 3), pad=(1, 1), num_filter = n_filter)
    net = mx.sym.BatchNorm(net, name='bnB'+str(i), fix_gamma=False)
    net = mx.sym.Activation(net, name='actB'+str(i), act_type='relu')
    net = mx.sym.Convolution(net, name='convB'+str(i), kernel=(3, 3), pad=(1, 1), num_filter = n_filter)
    net = net + identity # 直接加上之前的输出，即为残差结构

# 收尾工作
net = mx.sym.BatchNorm(net, name='bnFINAL', fix_gamma=False)
net = mx.sym.Activation(net, name='actFINAL', act_type='relu')
# 合并为1个通道
net = mx.sym.Convolution(net, name='convFINAL', kernel=(1, 1), num_filter=1)
net = mx.sym.Flatten(net)
# 输出为361个概率
net = mx.sym.SoftmaxOutput(net, name='softmax')

# 检查参数量
shape = {"data" : (32, 8, 19, 19)}
mx.viz.print_summary(symbol=net, shape=shape)
# 检查网络结构图
mx.viz.plot_network(symbol=net, shape=shape).view()
```

这里我们使用了6个残差模组，读者有兴趣可使用更深的网络，效果会更好，不过训练和运行时间也会更长。AlphaGo Zero的迷你版使用了20个残差模组，完整版使用了40个残差模组。

5.5.5 通道组合：Inception 模组

深度卷积网络的另一重要架构方法，是Google在ImageNet 2014竞赛中提出的Inception模组，它也在后续有诸多发展。

Inception的第1版来自《Going Deeper with Convolutions》[⊖]，它基于一个有趣的想法：不同大小的特征，可能需要不同大小的滤波器，因此我们不妨同时运用不同大小的滤波器，顺便再加上用滤波器无法实现的最大池化（步长需为1以保证最终得到的图像大小相等），如图5-31所示。

⊖ 地址为 <https://arxiv.org/abs/1409.4842>。

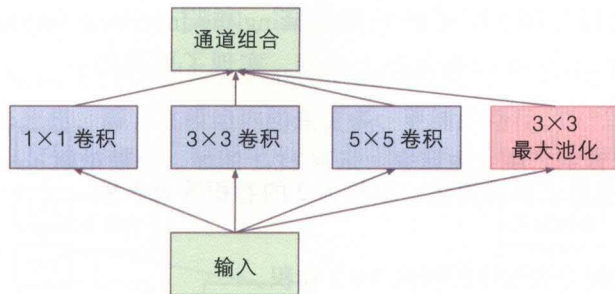


图 5-31 Inception v1 模组

注意，Inception 系列配图的网络运作方向是从下往上，且每个操作后都有非线性，在图中省略了。

举例说明这里的通道如何组合。假设本层的输入是 X ，将 X 通过 1×1 卷积生成了 A 张图像，再将 X 通过 3×3 卷积生成了 B 张图像，再将 X 通过 5×5 卷积生成了 C 张图像，再将 X 通过 3×3 最大池化生成了 D 张图像，如果这些图像的尺寸一致，就可以将它们组合 (concatenation) 在一起，成为 $A+B+C+D$ 张图像。

在 MXNet 实现上述架构的方法如下：

```
net = mx.symbol.Variable('data')

n1 = mx.symbol.Convolution(net, name='n1', kernel=(1,1), num_filter=32)
n1 = mx.symbol.Activation(n1, name='act1', act_type='relu')

n2 = mx.symbol.Convolution(net, name='n2', kernel=(3,3), pad=(1,1), num_filter=32)
n2 = mx.symbol.Activation(n2, name='act2', act_type='relu')

n3 = mx.symbol.Convolution(net, name='n3', kernel=(5,5), pad=(2,2), num_filter=32)
n3 = mx.symbol.Activation(n3, name='act3', act_type='relu')

n4 = mx.symbol.Pooling(net, name='n4', pool_type='max', kernel=(3,3), pad=(1,1),
    stride=(1,1))
n4 = mx.symbol.Activation(n4, name='act4', act_type='relu')

# 使用Concat组合所有通道
net = mx.symbol.Concat(n1, n2, n3, n4)
```

有经验的朋友会注意到，上述架构有一个问题，就是通道数会快速增长。因此在实际中，会加入 1×1 卷积作为瓶颈层，如图 5-32 所示。

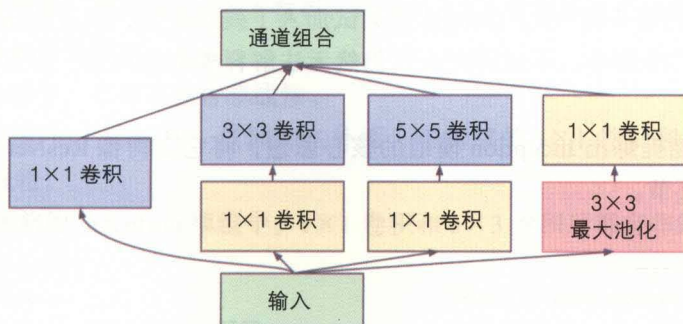


图 5-32 带瓶颈层的 Inception v1 模组

5.5.6 通道组合：Xception 架构，深度可分卷积

在 2016 年底，Keras 框架的创始人 Francois Chollet 提出了 Xception 架构^①，它运用了 Inception 模組的核心思想，即，在模組内构建更多、更细的路径。

首先，将 Inception 模組简化，如图 5-35 所示，先写成左图，注意到它等价于右图。

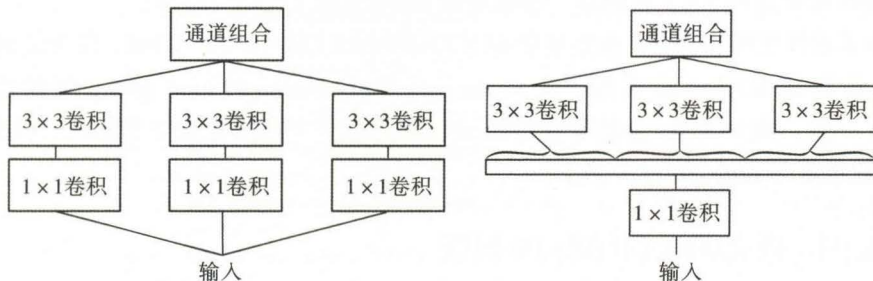


图 5-35 Inception 模組的简化

可见其中有 3 组路径，包含了 2 种思想（注意，网络运作方向是从下往上）：

- 先由 1×1 卷积负责处理通道间的混合。
- 再由 3×3 卷积负责处理图像内部的空间结构。注意 3×3 卷积也会在通道间做混合。

而 Xception 的想法是，引入更多的路径，甚至让每个通道都拥有 1 条独立路径，如图 5-36 所示。

这时的每个 3×3 卷积核都独立运作在各自的通道中，不再负责在通道间混合。这无疑可节省大量参数量，提高运算速度。此时，通道混合就完全由 1×1 卷积核负责。

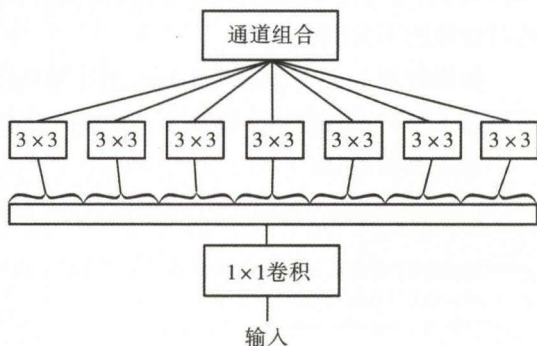


图 5-36 Xception 模組

在深度学习框架中，已有类似的操作，叫做深度可分卷积（depthwise separable convolution），不妨简称为可分卷积（separable convolution）。它与上图有两个区别：

- 在可分卷积中，是先执行 3×3 卷积，再执行 1×1 卷积。这个区别并不重要，因为我们会叠加很多层。
- 在可分卷积中，本身不包含非线性。
 - 如果我们加上非线性，只能加在外面，在内部的 1×1 卷积和 3×3 卷积之间没有非线性。
 - 而在此前的 Inception 模組中， 1×1 卷积和 3×3 卷积后都有非线性。

① 地址为 <https://arxiv.org/abs/1610.02357>。

经 Francois Chollet 的测试，可分卷积的效果更好，虽然它的非线性更少。这可能是因为在路径细分后，过多的非线性会阻碍信息的流动。

可分卷积还可节省参数量：

- ❑ 令输入通道数为 64，输出通道数为 128，忽略偏置。
- ❑ 如采用普通的 3×3 卷积层，参数量是 $64 \times 128 \times 3 \times 3 = 73728$ 。
- ❑ 如采用深度可分卷积，参数量是 $64 \times 3 \times 3 + 64 \times 128 \times 1 \times 1 = 8768$ ，仅为之前的 12%。

可分卷积在 MXNet 的实现，是将 `mx.sym.Convolution` 的 `num_group` 参数设置为与 `num_filter` 一致，则 MXNet 会将卷积神经元设置为每 1 个独立负责 1 个通道。注意输入数据的通道数也应与 `num_filter` 一致。

5.5.7 实例：再次训练 CIFAR-10 模型

在此采用 pre-act ResNet 结构再次训练 CIFAR-10 模型。这里的网络定义较为复杂，定义的要点是，对于每条通路，都应按照 Conv-BN-Act-Conv-BN-Act……顺序不断循环。

由于残差网络的训练较慢，为便于读者快速实验，这里使用了较小的网络，只有 1256400 个参数，比前文更少。同时，我们根据 Wide Residual Networks[⊖]的思想，使用了相对较宽的不太深的网络。

如果使用 `mx.viz.plot_network` 画出结构图，可看到这个网络的最深通路是 20 层，最浅通路是 5 层。即使在这个深度下，残差网络仍然拥有优势，可得到更高的准确率。

首先是初始化：

```
#-*-coding:utf-8-*-
```

```
import numpy as np
import logging
import mxnet as mx
```

```
logging.getLogger().setLevel(logging.DEBUG)
```

```
batch_size = 100 # 在此增大了批大小，因为ResNet的训练较慢
```

然后定义残差模块。

```
# 残差模块
```

```
def ResBlock(net, suffix, n_block, n_filter, stride=(1,1)):
    for i in range(0, n_block):
        if i == 0: # 注意第1个残差层的定义不同，读者可观察结构图思考原因
            net = mx.sym.BatchNorm(net, name='bn'+suffix+'_a'+str(i), fix_gamma=False)
            net = mx.sym.Activation(net, name='act'+suffix+'_a'+str(i), act_type='relu')
            # 对于第1个残差层，旁路从此开始
            pathway = mx.sym.Convolution(net, name="adj"+suffix, kernel=(1,1),
                                         stride=stride, num_filter=n_filter)
```

⊖ 地址为 <https://arxiv.org/pdf/1605.07146.pdf>。

```

# 回到主路
net = mx.sym.Convolution(net, name='conv'+suffix+'_a'+str(i),
    kernel=(3, 3), pad=(1, 1), num_filter = n_filter, stride=stride)
net = mx.sym.BatchNorm(net, name='bn'+suffix+'_b'+str(i), fix_gamma=False)
net = mx.sym.Activation(net, name='act'+suffix+'_b'+str(i), act_type='relu')
net = mx.sym.Convolution(net, name='conv'+suffix+'_b'+str(i),
    kernel=(3, 3), pad=(1, 1), num_filter = n_filter)
net = net + pathway # 加上旁路, 即为残差结构
else:
    pathway = net # 对于后续残差层, 旁路从此开始
    net = mx.sym.BatchNorm(net, name='bn'+suffix+'_a'+str(i), fix_gamma=False)
    net = mx.sym.Activation(net, name='act'+suffix+'_a'+str(i), act_type='relu')
    net = mx.sym.Convolution(net, name='conv'+suffix+'_a'+str(i),
        kernel=(3, 3), pad=(1, 1), num_filter = n_filter)
    net = mx.sym.BatchNorm(net, name='bn'+suffix+'_b'+str(i), fix_gamma=False)
    net = mx.sym.Activation(net, name='act'+suffix+'_b'+str(i), act_type='relu')
    net = mx.sym.Convolution(net, name='conv'+suffix+'_b'+str(i),
        kernel=(3, 3), pad=(1, 1), num_filter = n_filter)
    net = net + pathway # 加上旁路, 即为残差结构
return net

```

定义网络结构和模组:

```

net = mx.symbol.Variable('data')
# 为数据加上BN层可有一定的预处理效果
net = mx.sym.BatchNorm(net, name='bnPRE', fix_gamma=False)
# 将3*32*32变化为32*32*32
net = mx.sym.Convolution(net, name="convPRE", kernel=(3,3), pad=(1,1), num_filter=32)
# 将32*32*32变化为64*32*32
net = ResBlock(net, "1", 3, 64)
# 将64*32*32变化为64*16*16
net = ResBlock(net, "2", 3, 64, (2,2))
# 将64*16*16变化为128*8*8
net = ResBlock(net, "3", 3, 128, (2,2))
# 因为此前的最终层是卷积, 因此再做BN和Act处理
net = mx.sym.BatchNorm(net, name='bnFINAL', fix_gamma=False)
net = mx.sym.Activation(net, name='actFINAL', act_type='relu')
# 将128*8*8变化为128*1*1
net = mx.sym.Pooling(net, name="pool", global_pool=True, pool_type="avg",
    kernel=(1,1))
# 将128*1*1变换为128
net = mx.sym.Flatten(net, name="flatten")
# 将128变换为10
net = mx.sym.FullyConnected(net, num_hidden=10, name='fc')
net = mx.sym.SoftmaxOutput(net, name='softmax')

# 输出参数情况供参考
shape = {"data": (batch_size, 3, 28, 28)}
mx.viz.print_summary(symbol=net, shape=shape)

```



```
module = mx.mod.Module(symbol=net, context=mx.gpu(0)) # 网络模组
```

定义数据迭代器，并开始训练：

```
# 数据迭代器
train_iter = mx.io.ImageRecordIter(
    path_imgrec = "train.rec",
    data_shape = (3,28,28), # 图像通道和尺寸
    batch_size = batch_size,
    shuffle = True, # 开启随机次序
    rand_crop = True, # 开启随机裁剪
    rand_mirror = True, # 开启随机镜像
    random_h = 10, # 随机色相
    random_s = 20, # 随机饱和度
    random_l = 25, # 随机亮度
    max_random_scale = 1.20, # 随机放大
    min_random_scale = 0.88, # 随机缩小
    max_rotate_angle = 20, # 随机旋转
    max_aspect_ratio = 0.15, # 随机长宽比例
    max_shear_ratio = 0.10, # 随机倾斜比例
    fill_value = 0, # 四周填充黑色
)
val_iter = mx.io.ImageRecordIter(
    path_imgrec = "test.rec",
    data_shape = (3,28,28),
    batch_size = batch_size,
    shuffle = False,
    rand_crop = False,
    rand_mirror = False,
)
# 训练
module.fit(
    train_iter,
    eval_data=val_iter,
    initializer = mx.init.MSRAPrelu(slope=0.0), # 采用MSRAPrelu初始化
    optimizer = 'sgd',
    # 采用0.5的初始学习速率，并在每50000个样本后将学习速率缩减为之前的0.984倍
    optimizer_params = {'learning_rate' : 0.5, 'lr_scheduler' : mx.lr_scheduler.
        FactorScheduler(step=50000/batch_size, factor=0.984)},
    num_epoch = 2,
    batch_end_callback = mx.callback.Speedometer(batch_size, 50000/batch_size)
)
```

训练输出如下：

```
INFO:root:Epoch[0] Train-accuracy=0.401000
INFO:root:Epoch[0] Time cost=34.705
INFO:root:Epoch[0] Validation-accuracy=0.516600
INFO:root:Update[501]: Change learning rate to 4.92000e-01
INFO:root:Epoch[1] Train-accuracy=0.590100
INFO:root:Epoch[1] Time cost=34.918
INFO:root:Epoch[1] Validation-accuracy=0.669600
INFO:root:Update[1001]: Change learning rate to 4.84128e-01
.....
INFO:root:Update[98501]: Change learning rate to 2.08451e-02
```

```

INFO:root:Epoch[197] Train-accuracy=0.981720
INFO:root:Epoch[197] Time cost=35.876
INFO:root:Epoch[197] Validation-accuracy=0.930400
INFO:root:Update[99001]: Change learning rate to 2.05116e-02
INFO:root:Epoch[198] Train-accuracy=0.982380
INFO:root:Epoch[198] Time cost=35.341
INFO:root:Epoch[198] Validation-accuracy=0.928400
INFO:root:Update[99501]: Change learning rate to 2.01834e-02
INFO:root:Epoch[199] Train-accuracy=0.982080
INFO:root:Epoch[199] Time cost=35.183
INFO:root:Epoch[199] Validation-accuracy=0.930200

```

可见，虽然网络的参数更少，但最终准确率达到 93.0% 左右，比之前更高。读者可在此基础上实验更多架构和调参。

5.6 架构技巧：更多进展

5.6.1 残差网络进展：ResNext、Pyramid Net、DenseNet

ResNext (<https://arxiv.org/pdf/1611.05431.pdf>) 是残差思想与 Inception 思想的结合，也可看作 Xception 的前身，如图 5-37 所示。

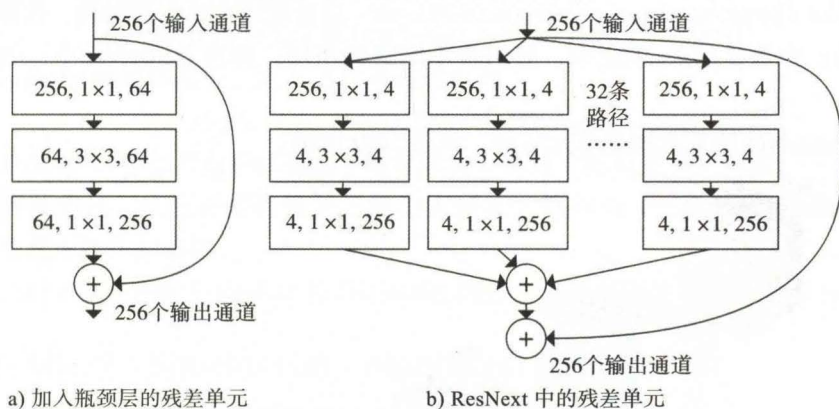


图 5-37 ResNext 模组

图中的单元按 A、N×N、B 格式，代表使用 N×N 卷积，输入为 A 个通道，输出为 B 个通道。

b 图运用了通道组合与路径拆分的思想，因此可在相似的计算量和参数量下采用整体更宽的层（例如 $4 \times 32 = 128$ ，大于 64），最终实现更佳性能。

Pyramid Net[⊖]的思想是使用越来越宽的金字塔形残差单元，如图 5-38 所示。

其中图 a 为普通的残差单元，图 b 为加入瓶颈层的残差单元，图 c 为较宽的残差单元

⊖ 地址为 <https://arxiv.org/abs/1610.02915>。

(即前文提到的 Wide Residual Networks^①，图 d 为 Pyramid Net。

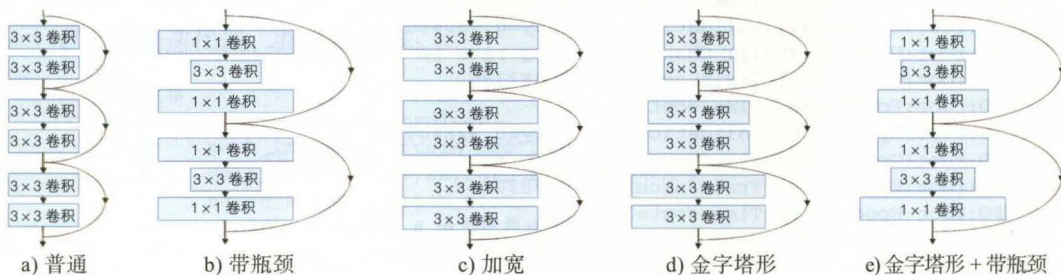


图 5-38 Pyramid Net 模组

这里的问题是，由于 Pyramid Net 中的通道数在不断增多，如果残差连接中的数据有 M 个通道，如何将其加到后续输出的具有 N 个通道的数据上（注意 $N > M$ ）。

如果使用 1×1 卷积将 M 个通道变为 N 个通道，会发现效果不佳，因为会阻碍信息传递。经研究人员实验，更好的方法是，将残差连接中的 M 个通道加到后续输出的前 M 个通道上。

最后，研究人员发现，将 pre-act 残差架构的第 1 个非线性层去掉，再在最后加入 1 个 BN 层，可能可实现更好的性能。这是深度学习的实验科学性的典型体现：研究人员会实验各种架构改变，有时会发现效果更好，但不一定能解释为什么效果更好。

DenseNet (<https://arxiv.org/pdf/1608.06993.pdf>) 是近期较为热门的架构，性能颇佳。其核心思想是：在传统的 ResNet 中，我们将不同路径相加；而在 DenseNet 中，我们将不同路径组合。

请看 DenseNet 的基本构成单元实例，如图 5-39 所示。

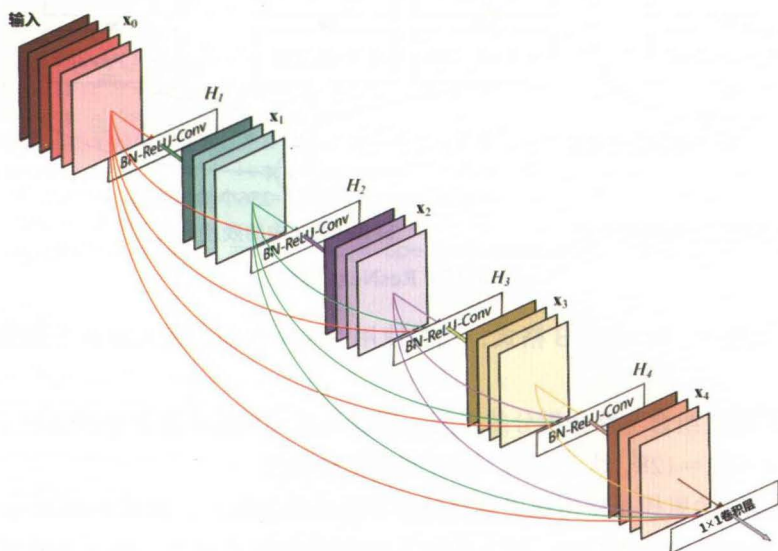


图 5-39 DenseNet 模组

① 地址为 <https://arxiv.org/pdf/1605.07146.pdf>。

这个单元的运作过程如下：

- 输入 x_0 是 5 个通道，经过 H_1 层的 BN-ReLU-Conv 操作，输出是具有 4 个通道的 x_1 。
- 将 x_0 和 x_1 组合为 $5+4=9$ 个通道，经过 H_2 层的 BN-ReLU-Conv 操作，输出是具有 4 个通道的 x_2 。
- 将 x_0 、 x_1 和 x_2 组合为 $5+4+4=13$ 个通道，经过 H_3 层的 BN-ReLU-Conv 操作，输出是具有 4 个通道的 x_3 。
- 以此类推。最终输出具有 $5+4+4+4+4=21$ 个通道。可见在 DenseNet 中的通道增长很快，因此可再经 1×1 卷积将通道数缩小。

最终网络的架构是多个单元模块的串联，如图 5-40 所示。

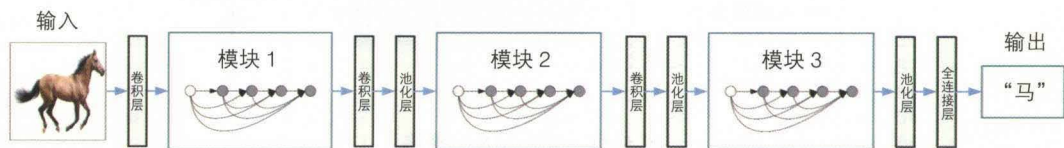


图 5-40 DenseNet 架构

在 DenseNet 中，每个中间层 H_n 使用很少的通道数，因为它的通道数增长很快。如果用数学表述，令输入为 x ，在使用 3 个中间层时，ResNet 的输出如下：

$$x + H_1(x) + H_2(x + H_1(x)) + H_3(x + H_1(x) + H_2(x + H_1(x)))$$

而 DenseNet 的输出如下，用 $[,]$ 代表通道组合：

$$[x, H_1(x), H_2(x, H_1(x)), H_3(x, H_1(x), H_2(x, H_1(x)))]$$

可见 DenseNet 将点对点加法换成了通道组合，有助于保留更多的信息。

关于残差架构，还有许多值得关注的论文研究，例如 FractalNet[⊖]可构建出如图 5-41 所示的相当复杂的残差架构。

Dual Path Net[⊗]类似于 ResNet 和 DenseNet 的混合，可用图像说明，如图 5-42 所示。

5.6.2 压缩网络：SqueezeNet、MobileNet、ShuffleNet

前文的主要关注点是提高网络的性能，但有时我们需要减少网络的参数量和计算量，例如在浏览器、手机端、嵌入式设备运行网络的时候。

这方面研究的起源是 SqueezeNet[⊕]，它可用 AlexNet 的五十分之一的参数量实现相似的性能，其核心组件类似 Inception 的架构，是 1×1 瓶颈层（称为“挤压”）和 1×1 、 3×3 卷积核（称为“扩展”）的组合，如图 5-43 所示。

⊖ 地址为 <https://arxiv.org/pdf/1605.07648.pdf>。

⊕ 地址为 <https://arxiv.org/pdf/1707.01629.pdf>。

⊗ 地址为 <https://arxiv.org/abs/1602.07360>。

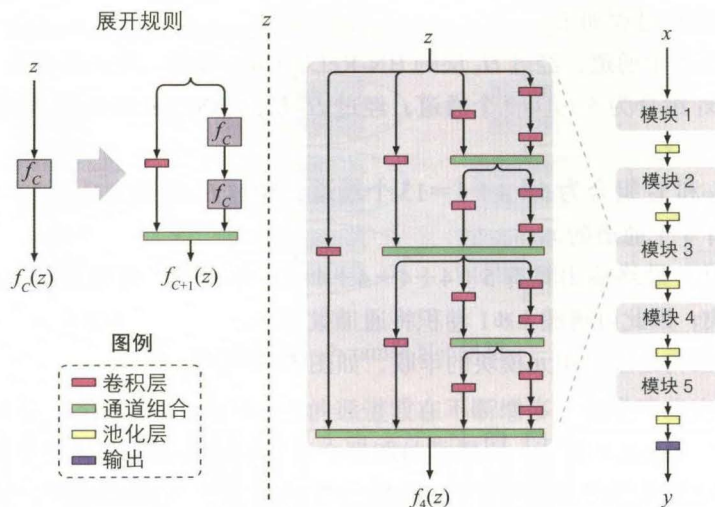


图 5-41 FractalNet 网络

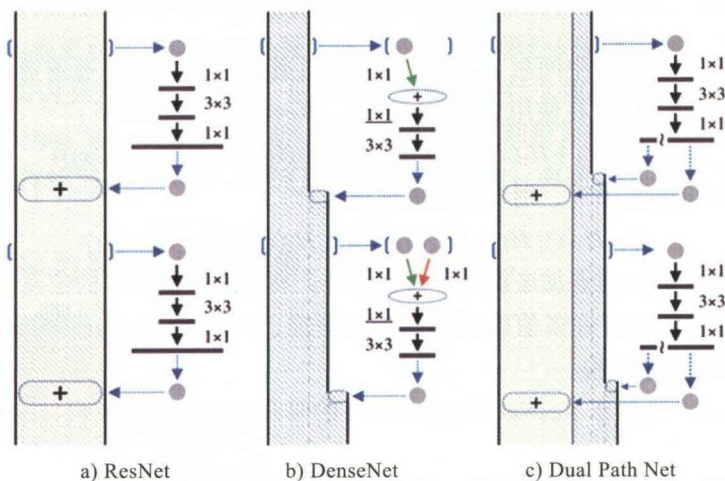


图 5-42 Dual Path Net 网络

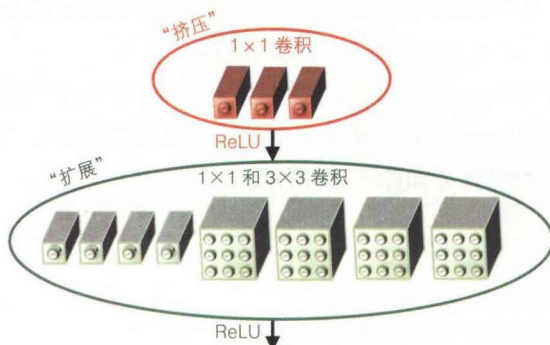


图 5-43 SqueezeNet 模组

但 SqueezeNet 的运行速度并未比 AlexNet 提高多少。因此 DarkNet 系列的作者提出了 Tiny DarkNet (<https://pjreddie.com/darknet/tiny-darknet/>), 比 SqueezeNet 更小、更快、更精确, 而且架构很简单(列表中为清晰起见, 省略 BN 和 Leaky ReLU 层):

层	滤波器数	大小/步长	输入	输出
0 conv	16	$3 \times 3 / 1$	$224 \times 224 \times 3$	$224 \times 224 \times 16$
1 max		$2 \times 2 / 2$	$224 \times 224 \times 16$	$112 \times 112 \times 16$
2 conv	32	$3 \times 3 / 1$	$112 \times 112 \times 16$	$112 \times 112 \times 32$
3 max		$2 \times 2 / 2$	$112 \times 112 \times 32$	$56 \times 56 \times 32$
4 conv	16	$1 \times 1 / 1$	$56 \times 56 \times 32$	$56 \times 56 \times 16$
5 conv	128	$3 \times 3 / 1$	$56 \times 56 \times 16$	$56 \times 56 \times 128$
6 conv	16	$1 \times 1 / 1$	$56 \times 56 \times 128$	$56 \times 56 \times 16$
7 conv	128	$3 \times 3 / 1$	$56 \times 56 \times 16$	$56 \times 56 \times 128$
8 max		$2 \times 2 / 2$	$56 \times 56 \times 128$	$28 \times 28 \times 128$
9 conv	32	$1 \times 1 / 1$	$28 \times 28 \times 128$	$28 \times 28 \times 32$
10 conv	256	$3 \times 3 / 1$	$28 \times 28 \times 32$	$28 \times 28 \times 256$
11 conv	32	$1 \times 1 / 1$	$28 \times 28 \times 256$	$28 \times 28 \times 32$
12 conv	256	$3 \times 3 / 1$	$28 \times 28 \times 32$	$28 \times 28 \times 256$
13 max		$2 \times 2 / 2$	$28 \times 28 \times 256$	$14 \times 14 \times 256$
14 conv	64	$1 \times 1 / 1$	$14 \times 14 \times 256$	$14 \times 14 \times 64$
15 conv	512	$3 \times 3 / 1$	$14 \times 14 \times 64$	$14 \times 14 \times 512$
16 conv	64	$1 \times 1 / 1$	$14 \times 14 \times 512$	$14 \times 14 \times 64$
17 conv	512	$3 \times 3 / 1$	$14 \times 14 \times 64$	$14 \times 14 \times 512$
18 conv	128	$1 \times 1 / 1$	$14 \times 14 \times 512$	$14 \times 14 \times 128$
19 conv	1000	$1 \times 1 / 1$	$14 \times 14 \times 128$	$14 \times 14 \times 1000$
20 avg			$14 \times 14 \times 1000$	1000
21 softmax				1000

另一种缩小参数大小的方法是压缩参数, 包括使用更低精度(如 16bit 甚至 8bit 的浮点数)的参数, 参数共享, 去除接近 0 的参数等。

通过特殊训练方法, 甚至可使用 1bit 的参数[⊖], 实现更小、更快的网络, 不过网络的性能也会明显降低。

在 2017 年的热门微型网络架构是 Google 提出的 MobileNet[⊖], 它的特点是使用了 Xception 中的深度可分卷积。如表 5-9 所示, 其中的 Conv 代表卷积, Conv dw 代表深度可分卷积, s 代表步长, FC 代表全连接层(为清晰起见, 省略 BN 和 ReLU 层), $K \times K \times A \times B$ 代表使用 $K \times K$ 卷积核, 输入为 A 个通道, 输出为 B 个通道。

表 5-9 MobileNet 架构

操作/步长	滤波器情况	输入尺寸
Conv/s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw/s1	$3 \times 3 \times 32dw$	$112 \times 112 \times 32$
Conv/s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw/s2	$3 \times 3 \times 64dw$	$112 \times 112 \times 64$
Conv/s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw/s1	$3 \times 3 \times 128dw$	$56 \times 56 \times 128$

⊖ 地址为 <https://arxiv.org/abs/1603.05279>。

⊖ 地址为 <https://arxiv.org/abs/1704.04861>。

(续)

操作 / 步长	滤波器情况	输入尺寸
Conv/s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw/s2	$3 \times 3 \times 128\text{dw}$	$56 \times 56 \times 128$
Conv/s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw/s1	$3 \times 3 \times 256\text{dw}$	$28 \times 28 \times 256$
Conv/s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw/s2	$3 \times 3 \times 256\text{dw}$	$28 \times 28 \times 256$
Conv/s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5 \times$ Conv dw/s1 Conv/s1	$3 \times 3 \times 512\text{dw}$ $1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$ $14 \times 14 \times 512$
Conv dw/s2	$3 \times 3 \times 512\text{dw}$	$14 \times 14 \times 512$
Conv/s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw/s2	$3 \times 3 \times 1024\text{dw}$	$7 \times 7 \times 1024$
Conv/s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
平均池化/s1	7×7 池化	$7 \times 7 \times 1024$
FC/s1	1024×1000	$1 \times 1 \times 1024$
Softmax/s1	输出	$1 \times 1 \times 1000$

而国内的旷视科技 (Face++) 提出了效率更高的 ShuffleNet[⊖]。在此前的各种 Inception 变种中, 每条路径是独立的, 相互之间的沟通不够。为此, ShuffleNet 加入了将通道打散 (channel shuffle) 的操作, 如图 5-44 所示。

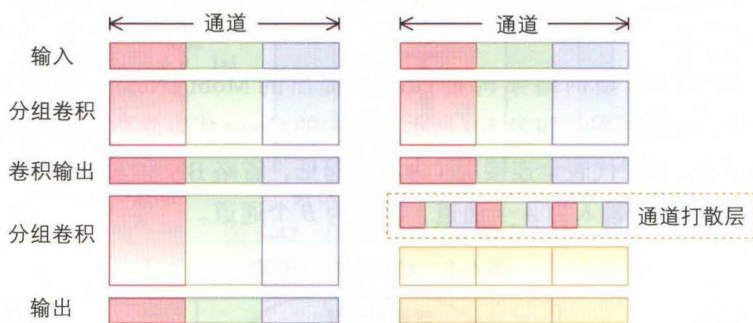


图 5-44 ShuffleNet 的思想

左图是一种原始的 Inception 架构, 具有 3 条完全独立的路径, 可用分 3 组的分组卷积实现。右图加入了通道打散层。例如, 假设输入是 3 条路径, 每条路径有 30 个通道, 那么可打散出如下的 3 条新路径:

- 将第 1 条路径的第 1~10 通道, 第 2 条路径的第 1~10 通道, 第 3 条路径的第 1~10 通道, 组合为新路径 1。

⊖ 地址为 <https://arxiv.org/pdf/1707.01083.pdf>。

- 将第1条路径的第10~20通道，第2条路径的第10~20通道，第3条路径的第10~20通道，组合为新路径2。
- 将第1条路径的第20~30通道，第2条路径的第20~30通道，第3条路径的第20~30通道，组合为新路径3。

于是每条新路径都包含了之前的3条路径的部分信息，有利于路径之间的信息流动，如图5-45所示。

ShuffleNet 中的典型单元，如图5-45中的b和c所示。

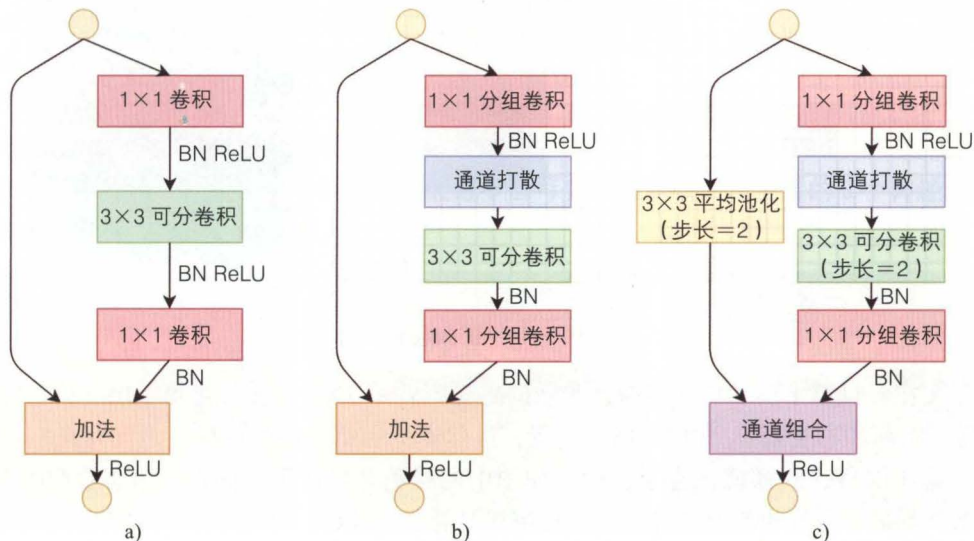


图 5-45 ShuffleNet 模组

- 图 a 是普通的采用可分卷积的残差单元。
- 图 b 是加入了通道打散层后的效果。由于通道打散层可在各组通道之间沟通信息，于是可将 1×1 卷积变为 1×1 分组卷积 (GConv)，即将通道分为几组，然后分别只在各组内进行 1×1 卷积，有利于减少参数数量和运算量。
- 图 c 是需缩小图像时的方法。假设输入的通道数为 N ，先将其进行步长为 2 的 3×3 平均池化，得到 N 个通道，再将其经过各种卷积和非线性操作得到 M 个通道，将两条路径的结果组合为 $N+M$ 个通道作为输出，这就比直接经过各种卷积和非线性操作得到 $N+M$ 个通道更为节省参数数量。

最终 ShuffleNet 可在 AlexNet 的二十分之一的运算量下实现相近性能，并可在 VGG 的三十分之一的运算量下实现相近性能。

在通道分组方面，目前最新的进展是 CondenseNet[⊖]，它在训练时让网络自动学会最佳的分组方法，可在 VGG 的五十分之一的运算量下实现相近性能，且实际在 ARM 嵌入式处理器的运行速度是 VGG 的 300 倍。

⊖ 地址为 <https://arxiv.org/abs/1711.09224>。

5.6.3 卷积核的变形

在现代深度卷积网络中往往只会使用 1×1 和 3×3 卷积核。不过，目前也有多种技巧让卷积核看到更宽广的区域。

首先是扩张卷积 (dilated convolution)^①，举例如图 5-46 所示，第 1 层的 3×3 卷积对应正常的 3×3 个点，第 2 层的 3×3 卷积对应每点之间隔了 1 个像素的 3×3 个点，第 3 层的 3×3 卷积对应每点之间隔了 3 个像素的 3×3 个点。

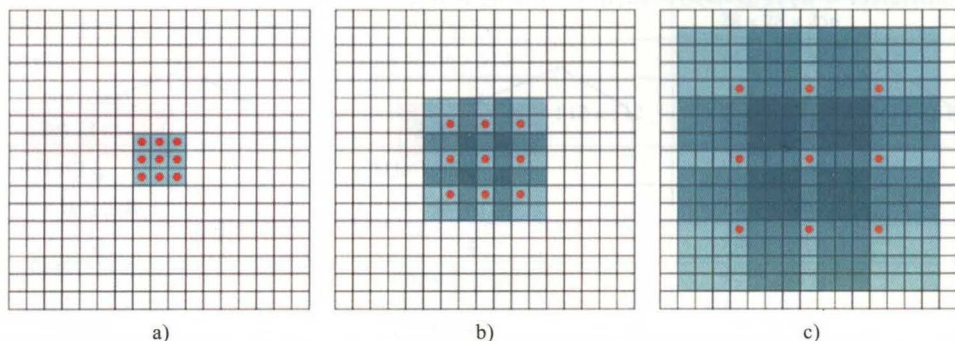


图 5-46 扩张卷积

那么在经过第 1 层和第 2 层后，实际的感受野就是 7×7 ；在经过第 1 层、第 2 层和第 3 层后，实际的感受野就是 15×15 。注意，虽然看上去第 2 层的采样点之间存在空隙，但在配合第 1 层后，采样就没有了空隙，如图中的彩色区域所示。同理，第 3 层在配合第 1 层和第 2 层后，采样也没有空隙。这是理解扩张卷积的关键。

而如果使用普通的 3×3 卷积叠加，使用 2 层只能实现 5×5 的感受野，使用 3 层只能实现 7×7 的感受野。可见扩张卷积能明显扩大感受野，它在图像分割、语音生成^②、机器翻译^③等领域都有应用。

然后是可变形卷积^④ (deformable convolution)。它希望让 3×3 卷积的采样点更为自由，并通过训练得到最佳的采样点，如图 5-47 所示。

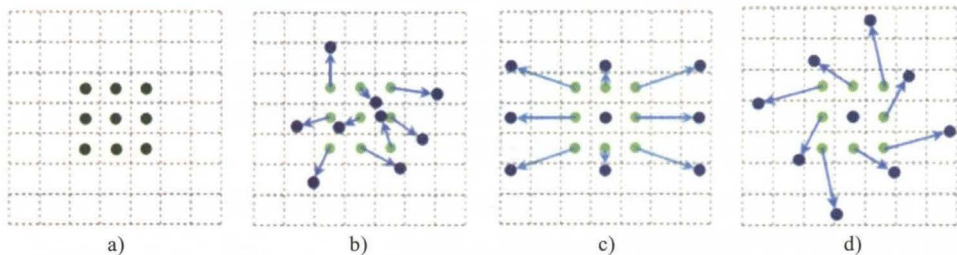


图 5-47 可变形卷积

① 地址为 <http://arxiv.org/abs/1511.07122>。

② 地址为 WaveNet, <https://arxiv.org/pdf/1609.03499.pdf>。

③ 地址为 ByteNet, <https://arxiv.org/abs/1610.10099>。

④ 地址为 <https://arxiv.org/abs/1703.06211>。

图 5-47a 是普通的 3×3 卷积的采样点，而图 b、c、d 是几种不同的采样方法。具体而言，会加入一个可训练的偏移层，为每个采样点找到最合适的位置。

可变形卷积核的有趣之处在于，它能让卷积核学会只看它所关心的图像区域，如图 5-48 所示。

图 a 是普通卷积层的叠加，只能按部就班观察正方形区域。图 b 是可变形卷积核的叠加，可学会观察不同形状的区域。

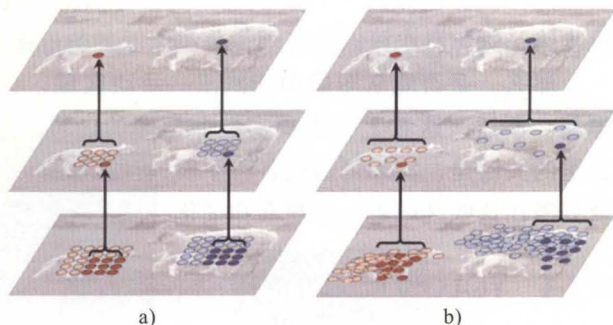


图 5-48 传统卷积与可变形卷积的对比

最终，网络中的每个神经元可将注意力集中到更明确的区域中。举例说明，经过 3 层可变形卷积核后，上层的 3 个神经元的最终采样点的情况，如图 5-49 所示。

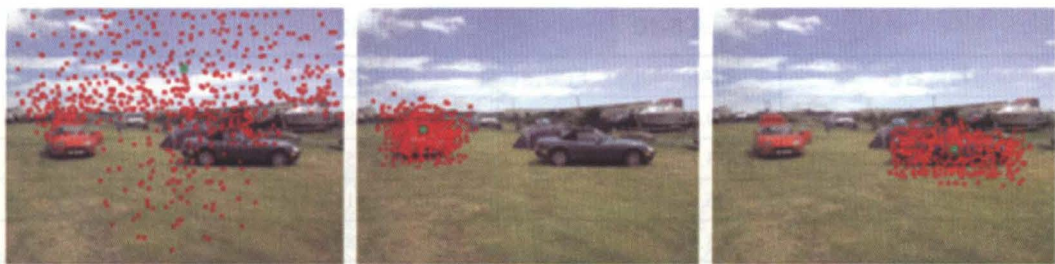


图 5-49 可变形卷积的效果

每个上层神经元有 $9 \times 9 \times 9 = 729$ 个采样点，用红点表示，可见，每个神经元的采样点都集中到了具有语义的图像位置上，这对于图像分割问题也会有帮助。

5.7 物体检测与图像分割

在解决图像分类后，下一步的问题就是物体检测和图像分割，如图 5-50 所示。它们对于自动驾驶、机器人等技术有重要意义。

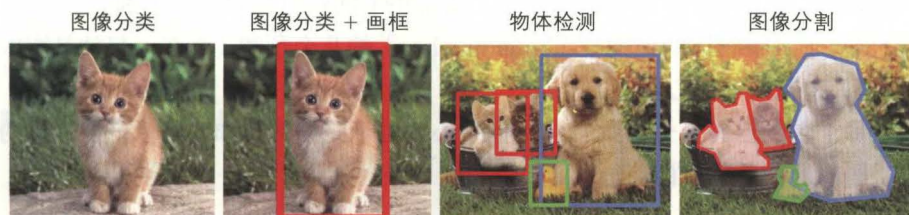


图 5-50 基本的图像问题

读者可能会想，前文我们能够可视化图像分类网络的判断依据，能否在此基础上直接输出图像的分割结果？目前来说，这种方法的精度太低，且图像分类的依据可能来自于其他物体，例如海水可以是将图像分类为船舶的依据之一，但海水并不是船舶的一部分。

那么，是否能从图像中切割出一块块区域，分别检测每个区域的分类结果？这是个思路。不过，这看上去需要实验非常多的区域，运行非常多次网络，因此速度可能堪忧。这里的关键是，实际可通过各种手段，预测关键区域的位置，这称为 region proposal。然后就只需在这些区域进一步观察和细化，可显著提升速度。

5.7.1 YOLO v1：实时的物体检测网络

YOLO 系列的作者是 Joseph Redmon，他也是此前介绍过的 DarkNet 系列的作者。这位作者很有幽默感，这里的 YOLO 是 You Only Look Once 的缩写，其实是故意为了和美国的一句网络流行语 You Only Live Once（“活在当下”）一致。

YOLO 系列和其他流行的物体检测网络的性能对比，如表 5-10 所示。

表 5-10 物体检测网络的性能比较

网络名称	mAP	FPS	网络名称	mAP	FPS
Fast R-CNN[5]	70.0	0.5	YOLOv2 288×288	69.0	91
Faster R-CNN VGG-16[15]	73.2	7	YOLOv2 352×352	73.7	81
Faster R-CNN ResNet[6]	76.4	5	YOLOv2 416×416	76.8	67
YOLO[14]	63.4	45	YOLOv2 480×480	77.8	59
SSD300[11]	74.3	46	YOLOv2 544×544	78.6	40
SSD500[11]	76.8	19			

其中 mAP 代表精度，FPS 代表速度，两者都是越高越好。上表是按架构的提出时间排序，可见 R-CNN 系列的精度高，但速度慢，而 YOLO 系列的速度非常快。在 YOLO v1 出现后，SSD 系列曾在综合性能上超过 YOLO v1，但 YOLO v2 又夺回了优势。

先看 2015 年的 YOLO v1[⊖]。它比此前的 R-CNN 系列快得多，是首个实现了实时物体检测的网络，其原理可概括如图 5-51 所示。

首先，将图像划分为 $S \times S$ 个网格。然后，对于每个网格，通过深度卷积网络：

- 给出对于其中物体所属类别的判断。图中用不同彩色代表不同类别。
- 给出对于其中物体的包围框（bounding box）的几个预测（画在一起就是图中的大量黑框）和判断的信心。
- 每个包围框有 5 个参数：包围框中心相对于网格的 x 和 y 坐标，包围框的长，包围框的宽，预测的信心。

举例说明：

⊖ 地址为 <https://arxiv.org/pdf/1506.02640.pdf>。

- 假设某个网格中是自行车轮的一部分，那么它会认为自己属于自行车的概率很高，并会给出对于整个自行车的所在位置的几个猜想。
- 假设某个网格中并没有什么东西，或者说属于背景，那么它给出的包围框的信心都会很低。



图 5-51 YOLO v1 的原理

最后，我们可综合考虑所有网格的意见，合并接近的框，给出物体检测结果。

对于经典的 Pascal VOC 数据集，其中有 20 个物体分类，因此网络会输出每个网格属于 20 个分类的概率。如果每个网格再输出 2 个包围框，就会再加上 $2 \times 5 = 10$ 个参数。因此最终每个网格需要输出 $20 + 10 = 30$ 个参数。

如果使用 7×7 网格，那么网络的最终输出就是 30 张 7×7 图像。网络架构如图 5-52 所示。

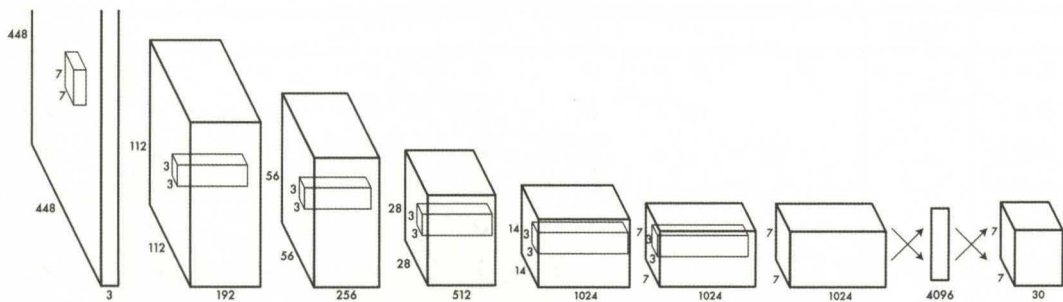


图 5-52 YOLO v1 的网络架构

实际训练中还有一些细节：

- 需要在 ImageNet 上预训练，这是常用的技巧，因为 Pascal VOC 数据集本身的数据量太少。
 - 这是一个多任务网络，需要为每个任务设置合适的损失函数，以保证最终效果。
- YOLO v1 的运行效果如图 5-53 所示。

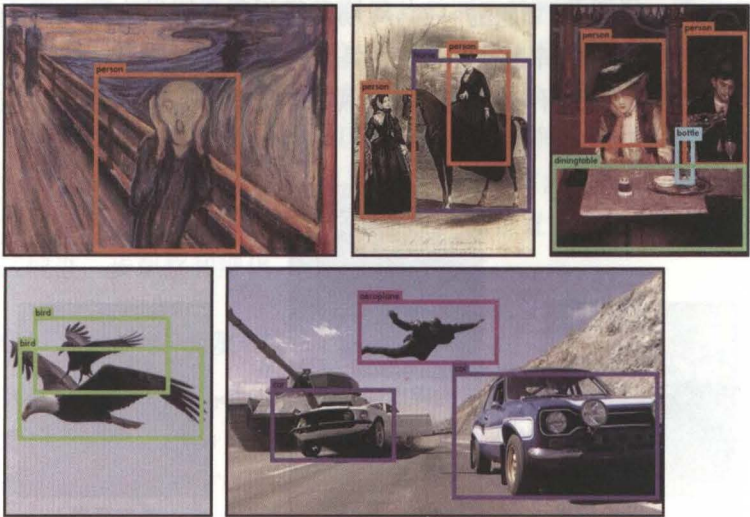


图 5-53 YOLO v1 的运行效果

YOLO v1 的最大优势是速度非常快，因为所有的结果都来自于同一个网络，是典型的端对端方法。不过它的识别精度相对较低，有时会漏掉小物体，有时会给出不够精确的包围框。后续的 YOLO v2 改进了这些缺点。

5.7.2 YOLO v2 : 更快、更强

YOLO v2 在 2016 年提出，论文为《YOLO 9000 : Better, Faster, Stronger》[⊖]。其中的 10 项改进，和在 VOC2007 数据集的 mAP 精度提升效果，如表 5-11 所示。

表 5-11 YOLO v2 的改进列表

	YOLO								YOLOv2
改进 1		✓	✓	✓	✓	✓	✓	✓	✓
改进 2			✓	✓	✓	✓	✓	✓	✓
改进 3				✓	✓	✓	✓	✓	✓
改进 4				✓	✓				
改进 5					✓	✓	✓	✓	✓
改进 6						✓	✓	✓	✓
改进 7						✓	✓	✓	✓
改进 8							✓	✓	✓
改进 9								✓	✓
改进 10									✓
	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	78.6

⊖ 地址为 <https://arxiv.org/abs/1612.08242>。

- 1) 加入 BN 层，训练速度更快，效果更好。
 - 2) 预训练时直接在更高分辨率的 ImageNet 上训练。
 - 3) 采用全卷积架构，去掉全连接层。
 - 4) 使用 5 种锚框 (anchor box)，其实就是 5 种固定比例的包围框，每个包围框可拥有各自的类别预测。这个思想来自于后文将介绍的 Faster R-CNN。这明显提高了对于小物体的召回率。
 - 5) 采用新的网络架构，即 DarkNet 系列。
 - 6) 包围框的比例，由预先训练数据集得到，而非人工设计，效果明显更好。训练时发现，应使用更多的瘦高的框，更少的扁长的框。
 - 7) 在预测包围框位置时，更倾向于预测中心点与当前网格中心接近的框。这可明显提高网络在训练时的稳定性。
 - 8) 加入类似残差架构的 passthrough 层，让网络可同时观察 13×13 网格和 26×26 网格上的特征信息。
 - 9) 在训练时采用多种不同大小的图像作为输入，提高网络的健壮性。
 - 10) 在运作时使用更高分辨率的图像作为输入，效果会更好（虽然速度也会更慢）。
- 最终 YOLO v2 战胜 SSD，夺回了“在给定的速度下准确率最高”的宝座。

而 Joseph Redmon 还不满足于此，在同一论文中还提出了 YOLO 9000 架构，它的特点是可预测 9000 多种物体分类。

具体方法是将训练图像分割的 COCO 数据集和 ImageNet 数据集，再嵌入 WordNet 语言数据库中的抽象概念树结构（例如“金毛寻回犬”属于“狗”，再属于“犬科”，再属于“动物”），并加入统计建模与条件概率的计算，如图 5-54 所示。

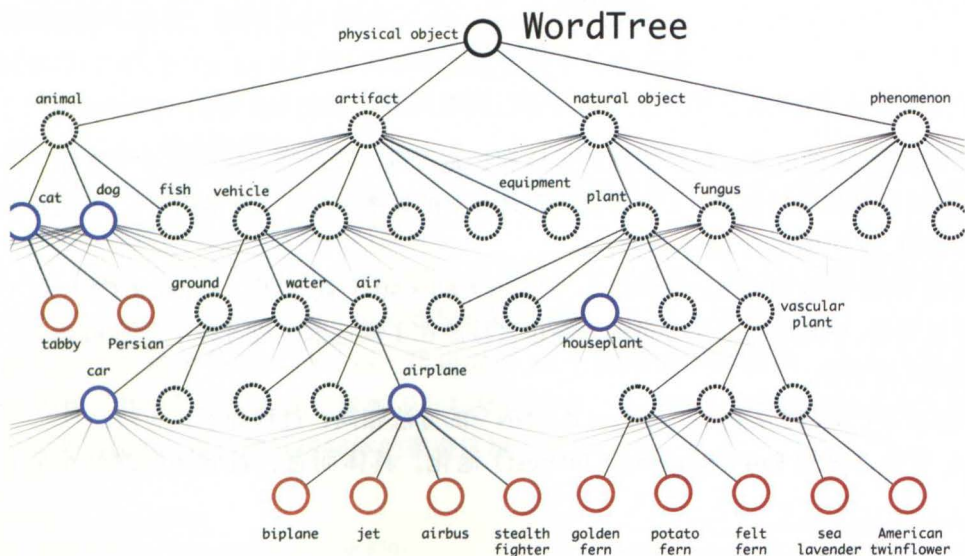


图 5-54 WordNet 语言数据库

5.7.3 Faster R-CNN：准确的物体检测网络

2015 年的 Faster R-CNN[⊖]是当时最准确的物体检测网络。在此我们也介绍其架构。

首先，训练一个图像分类网络（如 VGG-16），或采用现成的训练好的网络，然后将全连接层去掉，只考虑它的最后一个卷积层的输出。

例如对于 VGG-16，如果输入的图像是 $X \times Y$ ，最后一个卷积层的输出就是 $512 \times (X/16) \times (Y/16)$ ，因为其中会经过 4 次池化，每次缩小到原来的一半。

例如，若 $X=960$ ， $Y=640$ ，那么 VGG-16 的最后一个卷积层的输出就是 $512 \times 60 \times 40$ 。

然后在这个基础上训练一个 Region Proposal Network (RPN)，通过使用 3×3 和 1×1 卷积，将 $512 \times 60 \times 40$ 变为 $6k \times 60 \times 40$ ，其中 k 代表锚框的个数，一般取 $k=9$ ，为 3 种大小（例如 128, 256, 512）和 3 种比例（1:1, 1:2, 2:1）的组合，如图 5-55 所示。

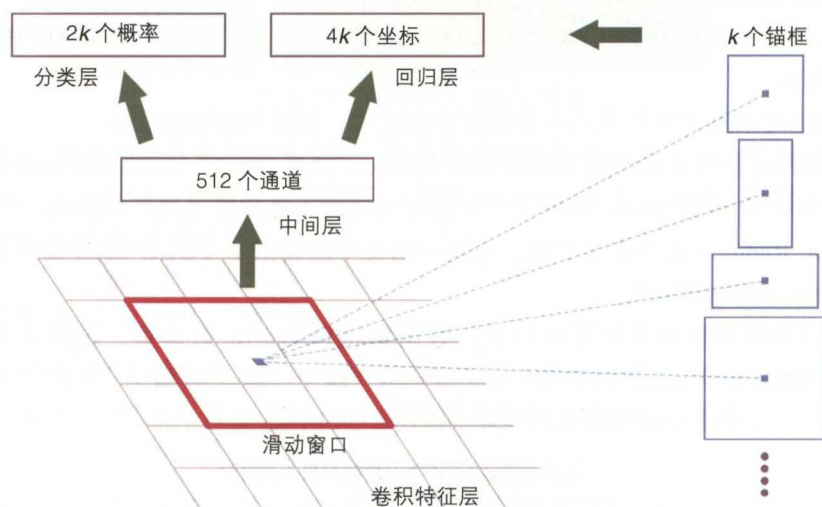


图 5-55 RPN 网络

在此说明 $6k$ 的由来。每个锚框的坐标和长宽可进一步改变，所以有 4 个坐标，再加上 1 个参数代表是物体的概率，1 个参数代表是背景的概率（这可提高准确率），因此每个锚框有 6 个参数， k 个锚框有 $6k$ 个参数。

因此 RPN 的输出是 $6k \times 60 \times 40$ ，对应 $k \times 60 \times 40$ 个候选框。Faster R-CNN 和 YOLO 的主要区别是，YOLO 把对于分类的判断也直接放在了候选框生成过程中，而 Faster R-CNN 还需做进一步分类。

Faster R-CNN 的分类方法与 Fast R-CNN[⊖]的方法相同，过程如下：

□ 首先，进行 RoI (Region of Interest) 池化。具体而言，若候选框的尺寸是 $P \times Q$ ，

⊖ 地址为 <https://arxiv.org/abs/1506.01497>。

⊖ 地址为 <https://arxiv.org/pdf/1504.08083.pdf>。

找到它所对应的网络最后一个卷积层的区域（如前所述，这个区域的尺寸是 $(P/16) \times (Q/16)$ ），将其纵横都切成 7 份，通过最大池化变为 $7 \times 7 \times$ 通道数。

□ 然后，将 $7 \times 7 \times$ 通道数，作为输入，经过一个小型全连接网络，输出候选框属于每个类别的概率，以及对于其位置的精细调整偏移值。

在得到所有候选框的分类信息后，就可合并所有框，得到最终的结果，如图 5-56 所示。

Faster R-CNN 的训练较为复杂，因为需同时训练 RPN 和最后用于分类的小型全连接网络，因此在 Faster R-CNN 的论文中提出了许多精细的训练技巧。

顺便一提，在最早的 R-CNN 中，需要用搜索给出候选框，而且在分类时也需要把每个区域在原图对应的区域都切出来。在整个分类网络中运行一遍，因此需要运行许多遍整个分类网络，速度特别慢。后续的 Fast R-CNN 解决了快速分类问题，而 Faster R-CNN 进一步用 RPN 代替了搜索。

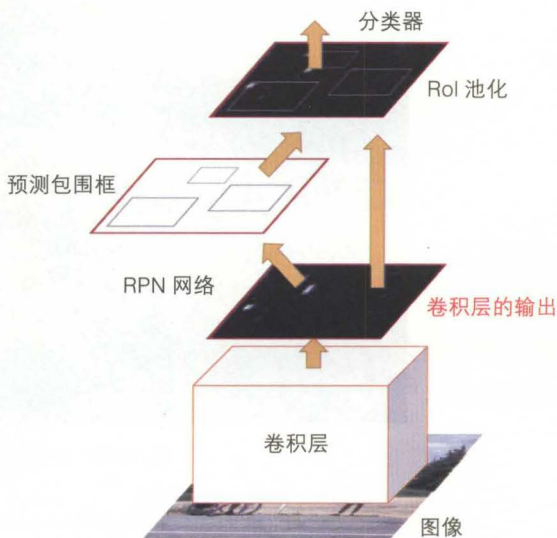


图 5-56 Faster R-CNN 的全架构

5.7.4 Mask-RCNN：准确的图像分割网络

2017 年的 Mask R-CNN^①在发表后倍受关注，因为它的效果很出色，能给出相当准确的物体的像素级边界，如图 5-57 所示。

Mask R-CNN 在 Faster R-CNN 的基础上主要做了几点改进：

1) 用 RoIAlign 代替 RoI 池化，具体是通过双线性插值得到更精确的结果，这对于后续的遮罩（mask）生成很重要。

2) 为卷积网络加入了 FPN 架构^②，它的思想是从图像的不同尺度提取特征，如图 5-58 所示，经实验在此的效果比残差网络更好。

3) 在 Faster R-CNN 在 RoI 后的分类 + 包围框分支的旁边，加入 1 个生成图像分割的遮罩的分支，它使用全卷积架构，包括卷积和转置卷积，如图 5-59 所示。

遮罩分支的输出是 $80 \times 28 \times 28$ ，代表对于 80 个物体的区域预测，每个预测是 1 张 28×28 图像，其中数字代表此位置属于此物体的概率。由于每个物体的预测是独立的，因此减少了类别竞争，可正确处理物体的重叠区域。

① 地址为 <https://arxiv.org/abs/1703.06870>。

② 地址为 <https://arxiv.org/pdf/1612.03144.pdf>。

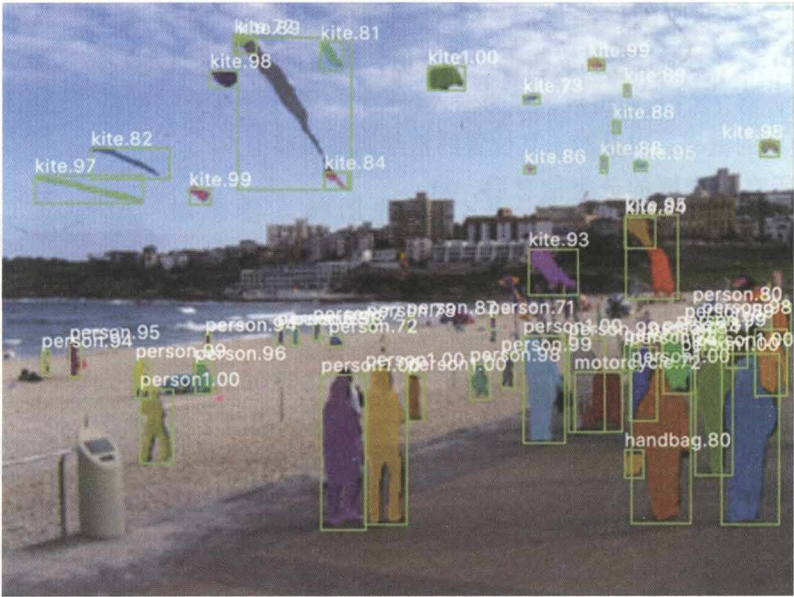


图 5-57 Mask R-CNN 的效果

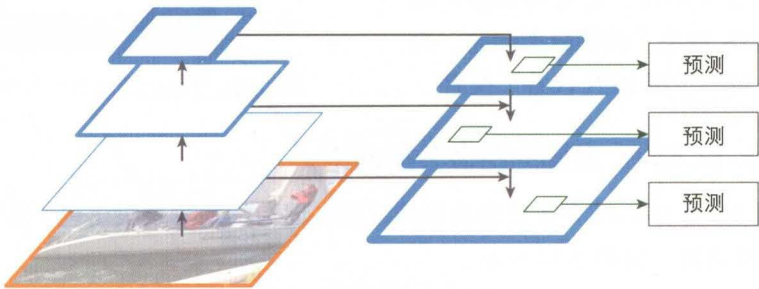


图 5-58 FPN 架构

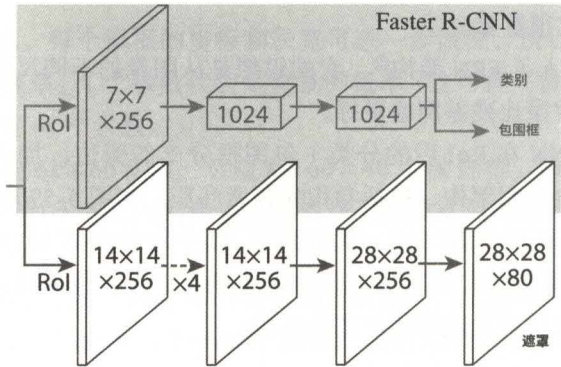


图 5-59 Mask R-CNN 中的遮罩分支

稍微修改遮罩分支，就可加入人体的姿态识别，如图 5-60 所示。



图 5-60 Mask R-CNN 实现的人体姿态识别

Mask R-CNN 还对 Faster R-CNN 做了许多细节改进，尤其是训练的细节改进，感兴趣的读者可阅读其论文。在 Mask R-CNN 之前的著名图像分割网络架构是 FCN[⊖]，也值得参考阅读。

5.8 风格转移

风格转移 (style transfer) 是一个有趣的话题，读者可能曾在手机 app (如 Prisma) 上看过它的效果。给定图像 A 和图像 B，它可得到图像 C，同时具有图像 A 的内容与图像 B 的风格。

例如，图 5-61 中将帆船的照片转变为水彩画的风格，但保留了帆船的内容。

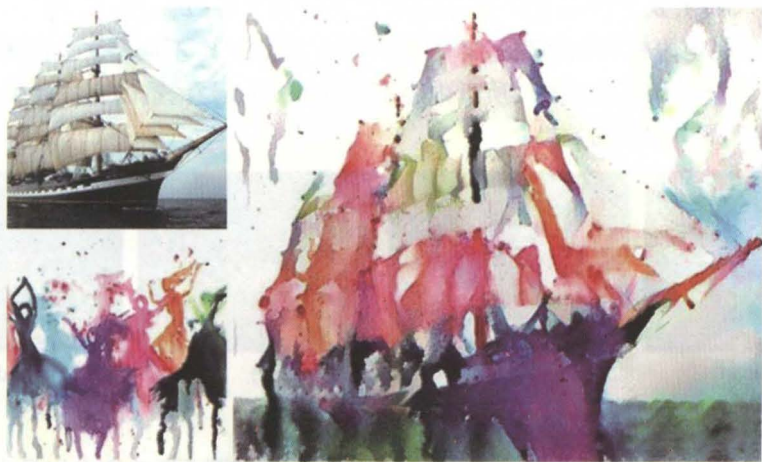


图 5-61 风格转移

[⊖] 地址为 <https://arxiv.org/abs/1411.4038v2>。

在 <https://github.com/ycjng/Neural-Style-Transfer-Papers> 列出了多篇风格转移的方法论文。其中经典的论文是《A Neural Algorithm of Artistic Style》^①，其思想如下：

- 图像 C 应保留图像 A 的内容，或者说图像 A 的语义。由于深度卷积网络特别擅长提取图像的语义，这就相当于要求图像 C 在深度卷积网络中的编码与图像 A 相近。
- 图像 C 应具有图像 B 的风格。这听上去很抽象，研究人员发现，可考虑特征间的 Gram 矩阵，类似于特征间的相关性。

具体而言，先选取在 ImageNet 上训练好的 VGG 网络。将图像 X 输入 VGG 网络后，第 l 层的输出有 N_l 个通道，每个通道有 M_l 个像素。定义 X_{ik}^l 为第 i 个通道在第 k 个像素上的值，则可定义 A 和 C 之间的内容损失为：

$$L_{\text{content}}(A, C, l) = \frac{1}{2} \sum_{ik} (A_{ik}^l - C_{ik}^l)^2$$

注意到内容损失是点对点的（ k 对 k ），因为我们希望保留图像上每个物体的位置。此时最小化内容损失，就相当于让 A 和 C 在语义内容上相近。选取不同的 l 会决定希望它们在低层语义上相似，还是在高层语义上相似。

而风格损失需要先计算 Gram 矩阵：

$$x_{ij}^l = \sum_k X_{ik}^l \cdot X_{jk}^l$$

然后风格损失的定义是：

$$L_{\text{style}}(B, C, l) = \frac{1}{4N_l^2 M_l^2} \sum_{ij} (b_{ij}^l - c_{ij}^l)^2$$

注意到风格损失是整体的，与像素 k 无关，因为风格是一个整体的概念。选取不同的 l 也会决定风格的层次。

最后可定义最终的损失为：

$$L(A, B, C) = \alpha (\sum v_l \cdot L_{\text{content}}(A, C, l)) + \beta (\sum w_l \cdot L_{\text{style}}(B, C, l))$$

其中 α 和 β 决定内容和风格之间的平衡，而 v_l 和 w_l 决定不同层次之间的平衡。举例，图 5-62 中图 a 是偏重风格的效果（所以内容不明显），图 d 是偏重内容的效果（所以风格较收敛）。图 b 和 c 是中间值的效果。



图 5-62 风格与内容的平衡

然后，选取 C 为一张随机噪声图像（或与 A 相同），再试图不断改变 C 以最小化 $L(A, B, C)$ ，最终就可得到一张包含了 A 的内容与 B 的风格 C。

① 地址为 <https://arxiv.org/abs/1508.06576>。

这个过程和通常的神经网络优化过程略有区别：

- 在通常的神经网络优化过程中，输入是固定的，我们希望调整网络的参数，以最小化输出的损失。
- 在风格转移中，网络的参数是固定的（是训练过的 VGG 网络），我们希望调整输入，以最小化输出的损失。后文在训练 GAN 时我们会遇到类似的问题，并看到如何在 MXNet 中实现这一点。

上述方法的缺点是，它的速度很慢，因为它需要不断逐步调整输入。

不过，由于 B 往往是固定的几张经过精心挑选的具有美观风格的图像，对于深度学习思想熟悉的读者就会想到，对于每个 B，可用 1 个深度网络直接学习最终的 C 和 A 之间的关系，直接将 A 变为具有 B 风格的 C。这就是《Perceptual Losses for Real-Time Style Transfer and Super-Resolution》[⊖]的思想。

对于风格转移的新进展可参阅《A Learned Representation For Artistic Style》[⊕]和《Deep Photo Style Transfer》[⊗]。例如，网络可成功将傍晚的照片变成深夜的照片，效果很真实，如图 5-63 所示。



图 5-63 风格转移的最新效果

最后，风格转移也可用 GAN 实现，效果有时更佳，具体请看后文的 CycleGAN 章节。

⊖ 地址为 <http://cs.stanford.edu/people/jcjohns/eccv16/>。

⊕ 地址为 <https://arxiv.org/pdf/1610.07629.pdf>。

⊗ 地址为 <https://arxiv.org/pdf/1703.07511v3.pdf>。

AlphaGo 架构综述

AlphaGo 系列的核心组件是策略网络与价值网络。对于棋盘上的任意一个局面，我们希望回答两个问题：

- 此时本方最佳的下一手是什么。这是策略网络希望回答的。
- 此时本方的胜率是多少（假设双方都完美地下完后续的棋），如图 6-1 所示。这是价值网络希望回答的。

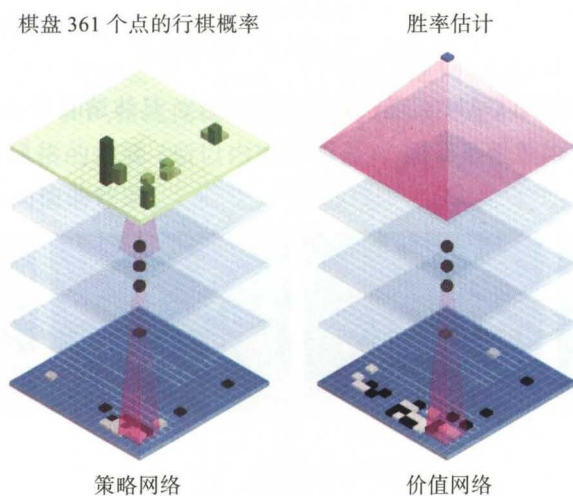


图 6-1 AlphaGo 的策略网络和价值网络

如果策略网络是完美的，每次都能直接告诉电脑最佳的着法，那么电脑就足以成为围棋之神。

如果价值网络是完美的，那么电脑也会成为围棋之神，因为最佳的策略，就是在每一步最大化自己的价值（胜率）。

然而围棋的策略问题和价值问题都很困难。如果读者看过围棋视频，会发现即使是围棋世界冠军们，对于下一手的选点也会有分歧，对于双方的形势孰优孰劣也可能会有不同见解。简而言之，策略和价值互为关联，但都难以直接得到答案。

原始 AlphaGo 的架构较为复杂，正是为了解决这个问题。它首先通过学习人类高手棋谱，得到初步的策略网络；再通过策略网络，得到价值网络；最后不断自我对弈，通过策略可以改善价值，通过价值可以改善策略，在不断循环中，AlphaGo 的棋力就越来越强。

而 AlphaGo Zero 通过更精心的设计架构和自我对弈学习的细节，去除了学习人类棋谱的过程，直接可从零开始不断循环进步，并达到更强的棋力。

6.1 从 AlphaGo 到 AlphaZero

AlphaGo 系列的论文可在 <https://github.com/B-C-WANG/AlphaGo-Zero-Paper> 下载。

6.1.1 AlphaGo v13 与 AlphaGo v18

让我们从 2016 年 1 月 Google DeepMind 团队在《Nature》上发表的著名论文说起。当时公布的 AlphaGo 版本为 2015 年 10 月的 AlphaGo v13，其架构可分为两大部分，即对弈与训练，如图 6-2 所示。

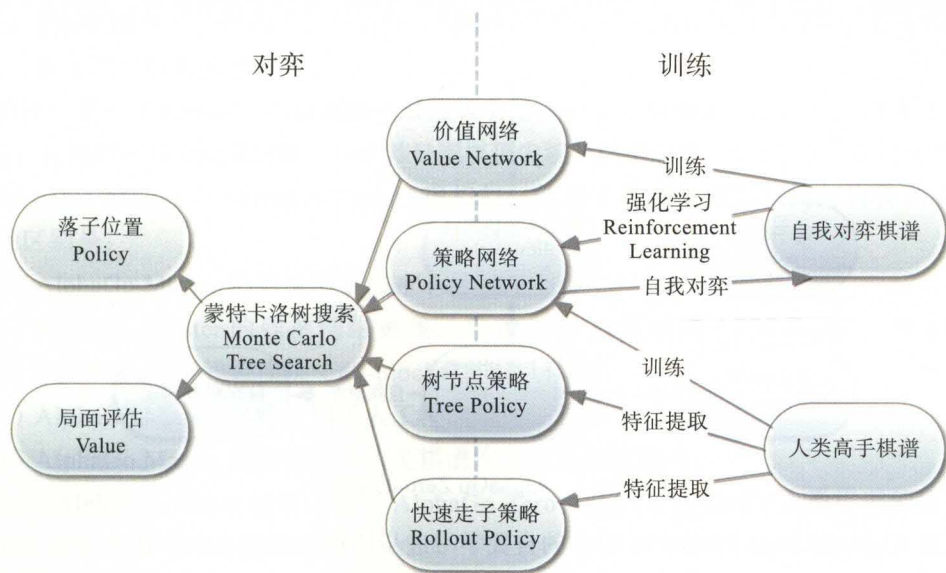


图 6-2 原始 AlphaGo 的架构

可将 AlphaGo v13 的运作过程总结如下：

- ❑ **初步训练**：通过学习人类高手棋谱，获得策略网络、树节点策略，快速走子策略。
- ❑ **强化训练**：通过自我对弈，进行深度强化学习，获得强化的策略网络，以及价值网络。由于自我对弈可不断进行，策略网络和价值网络会不断互为促进，电脑会越来越强，并且逐渐减少此前对于人类高手的模仿，树立自己的棋风。
- ❑ **对弈**：通过综合使用策略网络、价值网络、树节点策略、快速走子策略，提高蒙特卡洛树搜索的准确性，得到更为准确的落子位置和对于当前局面更为准确的评估。估值时，同时使用快速走子估值和价值网络估值，两者的权重各占 50%。如果局面很不利，例如预期胜率小于 20%，电脑会主动认输，这是围棋的礼节。

而在 2016 年 3 月与李世乭对弈的是 AlphaGo v18。它与 AlphaGo v13 相似，区别在于：

- ❑ 在 v18 中训练价值网络使用的自我对弈棋谱是由完整的蒙特卡洛树搜索生成。而在 v13 中，训练价值网络使用的自我对弈棋谱只是由策略网络生成，自我对弈速度更快，但棋力更低。
- ❑ 策略和价值网络使用了更大的神经网络，每层有 256 个卷积神经元。训练也持续了更长的时间。而在 v13 中，每层只有 192 个卷积神经元。
- ❑ 使用了 Google 研发的 TPU，显著加快训练和运行的速度。

6.1.2 AlphaGo Master 与 AlphaGo Zero

在 2017 年 10 月，Google DeepMind 团队在《Nature》上发布 AlphaGo Zero，它的架构更为简洁清晰，且完全脱离了人类棋谱，仅需自我对弈就能实现更强的棋力，如图 6-3 所示。

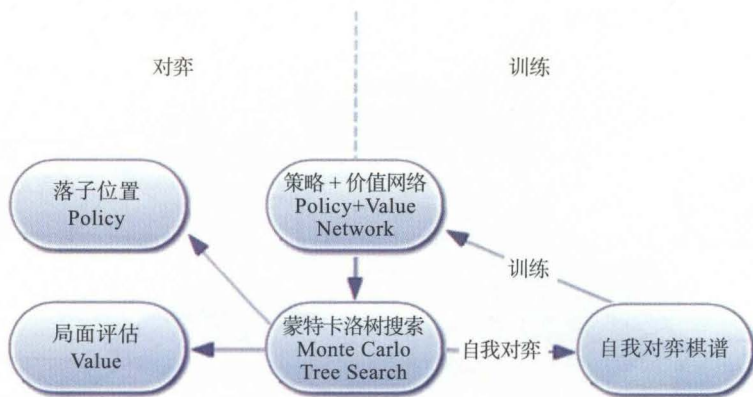


图 6-3 AlphaGo Zero 的架构

请看 AlphaGo Zero 20-blocks 版本的自我学习进步历程，如图 6-4 所示。

- ❑ 图 a 是 3 小时训练后，黑白双方都把棋下在棋盘中部，喜爱互相追杀，试图吃子，就像围棋的初学者。

- 图 b 是 19 小时训练后，它已知道要占角，占边（围棋的人类俗语是“金角，银边，草肚皮”），并展开激烈的争夺。
- 图 c 是 70 小时训练后，此时它的水平已超越人类，并且发现了许多新招法。

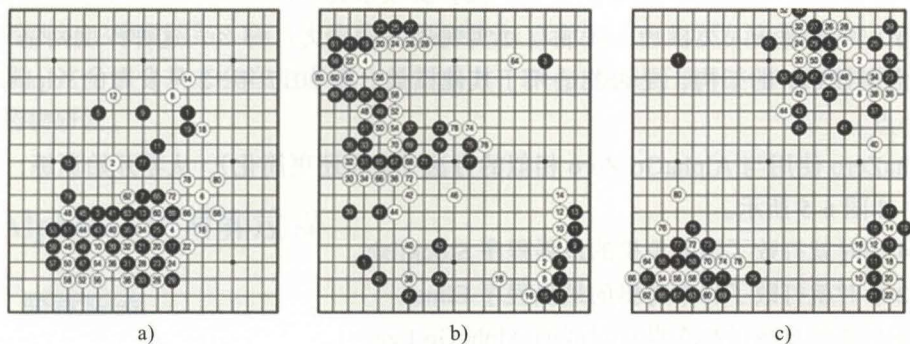


图 6-4 AlphaGo Zero 的进化

可将 AlphaGo Zero 的主要改进总结如下：

- 将策略与价值网络合二为一，成为同一个网络的两个分支。
 - 这属于多任务学习，增强了网络的健壮性，棋力显著更高。
- 使用残差网络架构。
- 网络的输入特征层减少到 17 个，无须使用此前版本中“数气”“征子”等人工构造的特征。
- 仍使用蒙特卡洛树搜索，但去除了其中的快速走子策略和树节点策略，仅使用策略 + 价值网络作为搜索的指引。

值得一提，在 AlphaGo Zero 之前的最强版本是 AlphaGo Master，它曾于 2017 年 1 月化名 Master，在围棋网站弈城和野狐，与中韩日围棋顶尖职业棋手对弈，取得 60 比 0 的辉煌战绩。

AlphaGo Master 与 AlphaGo Zero 已很相似，使用了类似的网络架构与自我训练方法。它们的区别是：

- 1) AlphaGo Master 仍然使用了快速走子估值。
- 2) AlphaGo Master 的残差网络更浅，只有 20 个残差模块（20-blocks）。而 AlphaGo Zero 的完全体有 40 个残差模块（40-blocks）。
- 3) AlphaGo Master 仍然使用了人工构造的特征层。
- 4) AlphaGo Master 仍然使用了人类棋谱作为初始训练输入。

为何 AlphaGo Master 的棋力低于 AlphaGo Zero，是因为它受到了人类棋谱的拖累吗？AlphaGo 团队并没有对此做测试。

根据笔者的经验，它们之间棋力差异的关键，应该是这里的第 1 点和第 2 点。当价值网络的准确度提高到一定程度后，快速走子估值就相当于噪声（我们在后文会看到它的方法很粗

略)，会干扰局势判断。同时，更深的网络，有助于对棋盘上复杂大龙的状况做出更准确判断。

6.1.3 解决一切棋类：AlphaZero

如果读者读过 AlphaGo Zero 的论文，就会感叹于其方法的简单：只需输入围棋规则，无须任何人类棋谱和人类棋理，就能自动实现超强的棋力。那么，这种深度网络 + MCTS（蒙特卡洛树搜索）的架构，是否能适用于其他棋类？在 2017 年 12 月发布的 AlphaZero 证实了这一点。

AlphaZero 使用与 AlphaGo Zero 相似的方法（实际还更简化），从零开始训练，棋力进展神速，如图 6-5 所示。

- 4 小时就打败了国际象棋的最强程序 Stockfish。
- 2 小时就打败了日本将棋的最强程序 Elmo。
- 8 小时就打败了与李世石对战的 AlphaGo Lee。

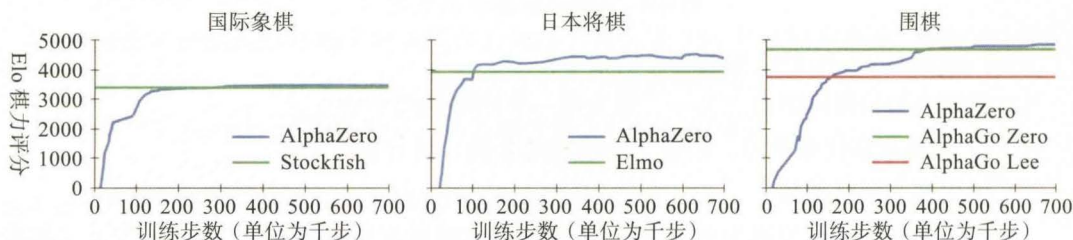


图 6-5 AlphaZero 的进化

这震动了国际象棋界，因为此前大家认为国际象棋与围棋的规则和策略相差甚远，因此很难想象它们可用相似的方法解决。而且 Stockfish 并不是一个简单的对手，它的棋力早已远远超越所有人类国际象棋大师，它的代码中有无数人类精心构造的算法技巧，但 AlphaZero 完全没有人类输入。

然而，在 AlphaZero 眼中，各种棋类并没有区别，它都可以轻松学会。Stockfish 就像一位武学大师，碰上了用枪的 AlphaZero。在 AlphaZero 和 Stockfish 的 100 盘对局中，AlphaZero 无一败绩。

国际象棋大师在研究 AlphaZero 和 Stockfish 的对战棋谱后表示，AlphaZero 的棋风与基于 Alpha-Beta 搜索的 Stockfish 等传统 AI 有明显区别，更富有策略性，弃子灵活坚决，更能掌控全局，深刻地谋划 (maneuver)：

- Stockfish 作为传统 AI 的代表，有人类设计的评估函数，整天想着“怎样可在多少步后吃对方的子”。
- AlphaZero 是“以德服人”，既然 Stockfish 贪吃就给 Stockfish 吃，没关系，一切尽在 AlphaZero 的掌握之中，它甚至能让 Stockfish 自己把自己的棋子堵在角落里出不来，成为废子。这正是价值网络的典型威力体现。

□ AlphaZero 牢牢占据主动权，Stockfish 疲于奔命，四处救火。

需要指出，训练 AlphaZero 所需的计算资源也是海量的，需要 5000 个 TPU 用于生成自对弈棋谱，相当于上万张 GTX 1080 Ti。不过，随着硬件的发展，这样的计算资源会越来越普及。未来的 AI 会有多强大，确实值得思考。

笔者认为，深度网络 + MCTS 是非常强的组合，因为 MCTS 可为深度网络补充逻辑性。笔者预测，这个组合未来会在更多场合显示威力。DeepMind 可能下一步会将其用于蛋白质折叠和药物研发。

6.2 AlphaGo 的对弈过程

6.2.1 策略网络

以 AlphaGo v13 为例，策略网络的架构如图 6-6 所示。其中第 1 到 12 层后的非线性都为 ReLU。

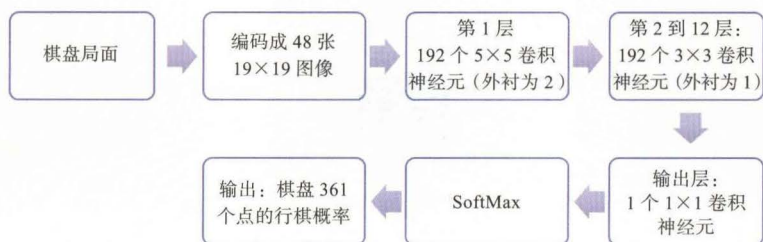


图 6-6 原始 AlphaGo 中的策略网络

让我们假设电脑持黑棋先行，从空白的 19 路棋盘开始，共有 $19 \times 19 = 361$ 个可选点（在此把对称的点也认为是不同的点），如图 6-7 所示。

面对茫茫棋盘，如何落下第 1 手？传统电脑博弈程序的做法，是创建开局库（opening book），把常见的开局套路存储起来，随后即可按图索骥。

然而，围棋的变化实在太多，开局库只能覆盖棋局前期的常见变化。当电脑遇到没有见过的套路，就需要从头搜索，这往往意味着棋力大减。更重要的是，开局库会成为电脑进步的枷锁，限制电脑发现比人类更好的开局。

对于此，AlphaGo 给出了新答案：可通过深度卷积网络，直接推荐下一手的着法。这就是策略网络。策略网络的输出是 361 个数字，标志着下一手选择棋盘上的 361 个点的概率。

请看我们将在本书训练的策略网络运行实例。为方便说明，图 6-8 中只列出了概率最高的 9 个点，用 1~9 代表策略网络对于下一手的前 9 位推荐。

可见，对于空白的棋盘，黑棋的第 1 手，最推荐的着法是图 6-8 中的 1 号点，星位 (Q16)；其次推荐的着法是图中的 2 号点，小目 (R16)；等等。

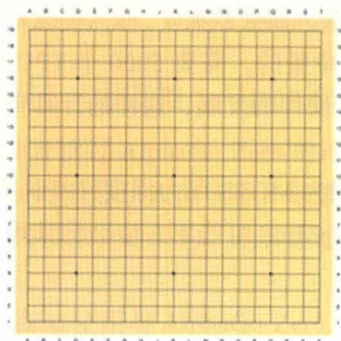


图 6-7 围棋的 19×19 空棋盘

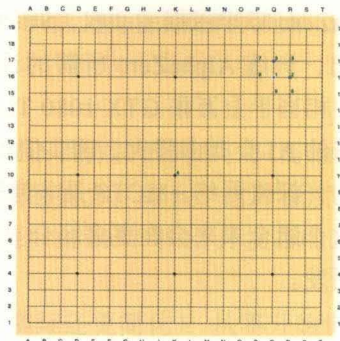


图 6-8 策略网络输入空棋盘后的输出

这听上去有点像开局库，但策略网络与开局库有重要区别：对于任意局面，策略网络都能给出不错的选点，宛如真正掌握了围棋的要领，如图 6-9 所示。

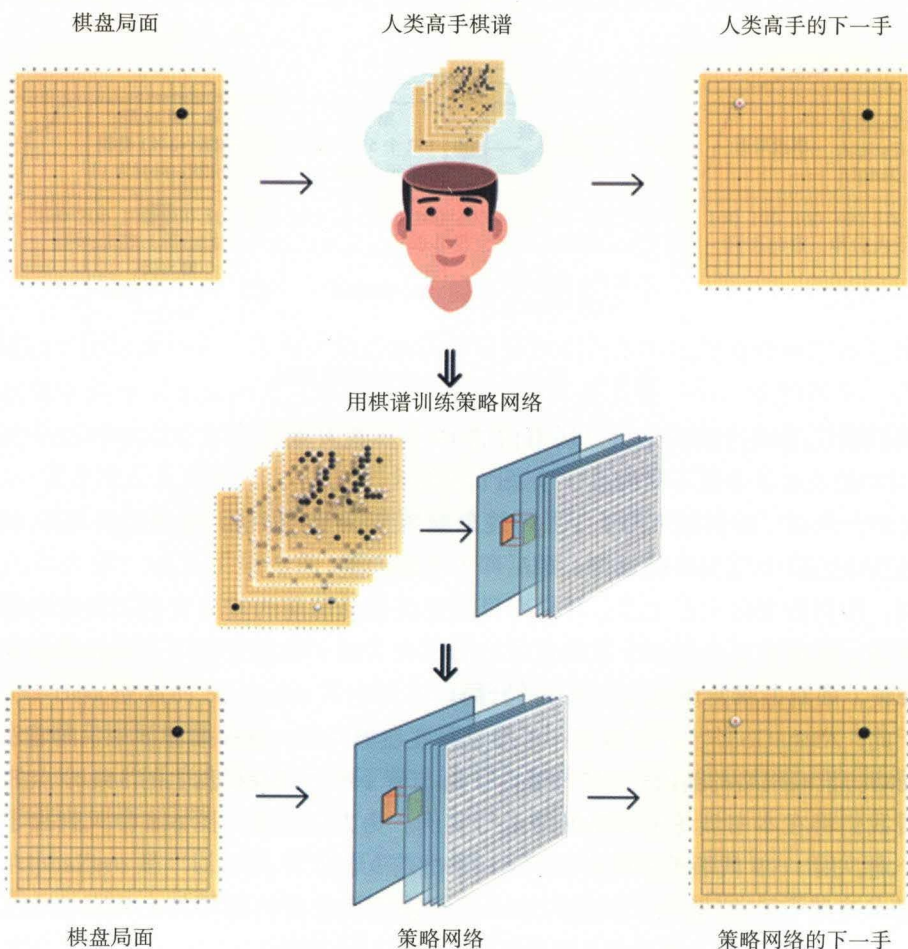


图 6-9 策略网络的学习过程

如前所述, 神经网络的神奇在于泛化能力, 它可从学习的棋谱中总结出更高层次的规律, 而不是死记硬背, 因此即使面对全新的局面, 也往往能给出不错的答案。

在这个意义上, 我们可把策略网络看作一本动态的棋书, 并且它还能通过自我对弈不断进步, 正如人类棋手可自己复盘, 演练后续变化, 从中找到更佳的下法。

根据策略网络的推荐选点, 就已经可以进行对弈。例如, 如果每次都选我们策略网络输出的概率最高的选点, 将得到如图 6-10 所示的棋局。

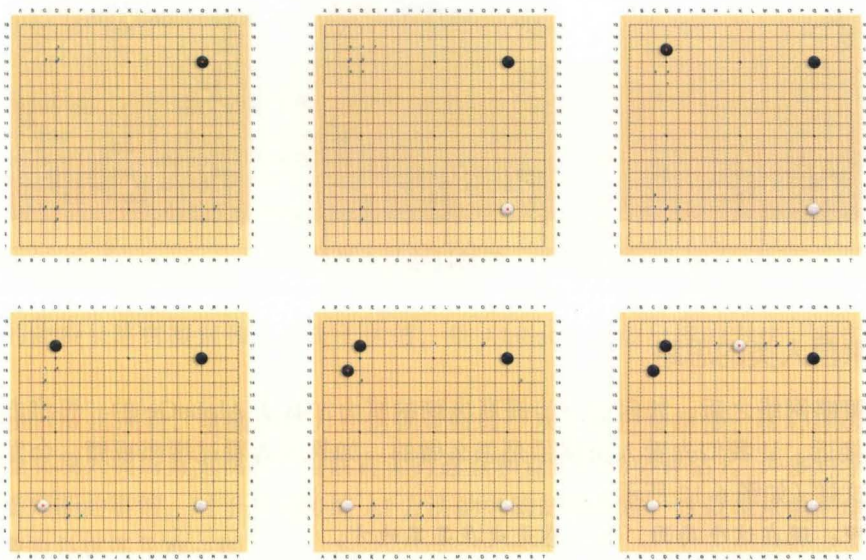


图 6-10 策略网络自我对弈

可见, 我们的策略网络很稳健, 黑棋选择了传统的“星·无忧角”布局, 白棋第 6 步落在黑棋的 3 颗子间, 也是经典的手法。这与训练时所采用的棋谱的风格有关。

策略网络的棋风来自于它所学习的棋谱, 所以如果训练时给它不同的棋谱, 它就会学到不同的棋风。例如, AlphaGo 通过自我对弈, 发现布局时“大飞守角”和“二间高守角”比“无忧角”的最终胜率更高, 于是策略网络就会更多地运用这些布局。

目前最高水准的棋谱, 莫过于 AlphaGo Zero 的自我对弈棋谱。如果我们能得到大量 AlphaGo Zero 的自我对弈棋谱, 仅用策略网络就可能达到职业级的棋力。遗憾的是 Google DeepMind 只选择公布了其中的少量对局。

在某种意义上, 深度学习的灵魂在于数据。高质量的学习数据尤为重要, 也是各家公司的保密对象。

言归正传。如果我们仅仅使用人类高手棋谱训练策略网络, 只需几天的训练, 电脑就能拥有约奕城 1 段的棋力, 特别是大局观相当出色, 达到奕城 3 段左右的水准。这是许多人需要学习几年才能达到的高度, 更是此前的电脑围棋程序运用各种复杂技巧才能达到的高度。

而这一切不需要训练者对围棋有任何了解。可以说，深度学习将许多复杂的问题变得更大众化，更简单了。只需准备足够充分的数据，再实验各种网络架构和训练方法，即可自动训练出相当不错的模型。

请看我们的策略网络与某位野狐 3 段的实战快棋对局。白棋是策略网络，取得中盘胜。前 150 手如图 6-11 所示。



图 6-11 策略网络与人类的对局

6.2.2 来自人类的思路

有了策略网络之后，如何进一步提高电脑的棋力？其实 AlphaGo 的“思维方式”和人类棋手很接近。让我们先看人类棋手是怎么做的。如果读者看过围棋视频，会发现讲棋过程常常是这样（如图 6-12 所示）：

解说甲：现在这个局面，第一感是走在某某地方。

解说甲：（摆一下后续的变化图）如果双方这样走完，现在局势如何如何。

解说乙：如果这一步走在这里呢？

解说甲：这个看上去是好棋 / 不是好棋啊（摆一下新的变化图）。如果这样走完，双方局势是如何如何，看来这步可行 / 不可行。

由此可见，直觉（或者说“棋感”）在围棋中很重要。首要的是直觉的选点：

- ❑ 围棋的棋盘很大，所以初学者常常会觉得到处都可以下，茫然不知所措，或经过长时间思考仍然下出坏棋。
- ❑ 职业棋手能迅速将可能的着手缩小到几手（甚至“只此一手”“定式”），从而可全神贯注分析它们的后续变化，因此下得又快又准。

但是，直觉的选点不一定永远准确，就像“定式”不能死记硬背，还需用后续的“变化图”和“局势评估”检验：

- ❑ 有时直觉会误导棋手的判断，让棋手下出“随手棋”。



图 6-12 围棋的人类讲解过程

□ 有时下一步的正解很隐蔽，是棋手的“盲区”。

人类的直觉选点，对应 AlphaGo 的策略网络。所以为了提高棋力，需要引入变化图和局势评估，这两者恰恰可以对应 AlphaGo 的蒙特卡罗树搜索和价值网络。

AlphaGo 的各大组件可与人类棋手的思维一一对应。围棋 AI 的发展史，其实也是与人类思维逐渐靠近的过程，如表 6-1 所示。

表 6-1 围棋 AI 和人类思维的对比

人类棋手	早期围棋程序（如《手谈》）	AlphaGo 前的程序（如 CrazyStone）	AlphaGo v18	AlphaGo Zero
最高棋力：职业九段	最高棋力：业余 1 级	最高棋力：业余 4 段	最高棋力：相当于职业十段	最高棋力：相当于职业十二段以上
棋感：直觉选点	人工定义的简单函数	线性的特征匹配	策略网络	综合网络
棋感：局势评估	人工定义的简单函数	快速走子估值	价值网络 + 快速走子估值	综合网络
对后续变化图的思考	简单的树搜索	蒙特卡洛树搜索	蒙特卡洛树搜索	蒙特卡洛树搜索

表中的蒙特卡洛树搜索和快速走子估值，是 AlphaGo 之前出现的方法，我们会在后文介绍。

在 AlphaGo 出现前，电脑的棋力在达到业余段位水准后进步缓慢，核心原因是无法快速且准确地实现人类的直觉棋感（包括直觉的选点和直觉的局势评估）。许多对于人类高手是一眼就能判断的事情，对于采用传统 AI 方法的电脑却很困难。这就像从前的电脑很难学会判断图像中是猫还是狗一样。

而 AlphaGo 的强大，首先在于运用深度卷积网络学会了人类棋手的棋感，然后通过自我对弈，进一步学习了远远比人类所有棋手平生所见更多的海量局面，最终比人类棋手的棋感更加准确可靠。

然后 AlphaGo 再加上蒙特卡洛树搜索，形成了完善的组合。由于电脑可以不知疲惫地搜索，因此如果电脑的棋感全面达到和超越人类，人类棋手就将像与火箭赛跑一般，再也难有胜算。

目前而言，如第 1 章所述，深度神经网络相对较弱的是逻辑性。因此，许多电脑围棋 AI 在引入深度学习后，是在看似虚无缥缈的大局观和形势判断方面超过人类，反而是在某些直线的、有明确最佳答案的选点和估值上（如某些死活，官子）相对较弱。

另一方面，由于深度神经网络的棋感会随着训练越来越强，人类棋手会越来越难将棋局引入电脑不擅长的领域。因此如果人类面对 AlphaGo Master 和 AlphaGo Zero 这样经过长期训练的对手，确实几乎不可能找到胜机。

6.2.3 蒙特卡洛树搜索与估值问题

蒙特卡洛树搜索（Monte Carlo Tree Search, MCTS）是现代博弈程序的基础算法。即使

是强大的 AlphaGo Zero 和 AlphaZero，也仍然需要 MCTS，不能纯靠深度卷积网络。

在此简单说明其工作过程。先看树搜索（tree search），电脑可将当前局面下的所有着法按策略网络给出的概率排序，就像人对于每一步的后续变化的思考，称为搜索树（search tree），如图 6-13 所示。

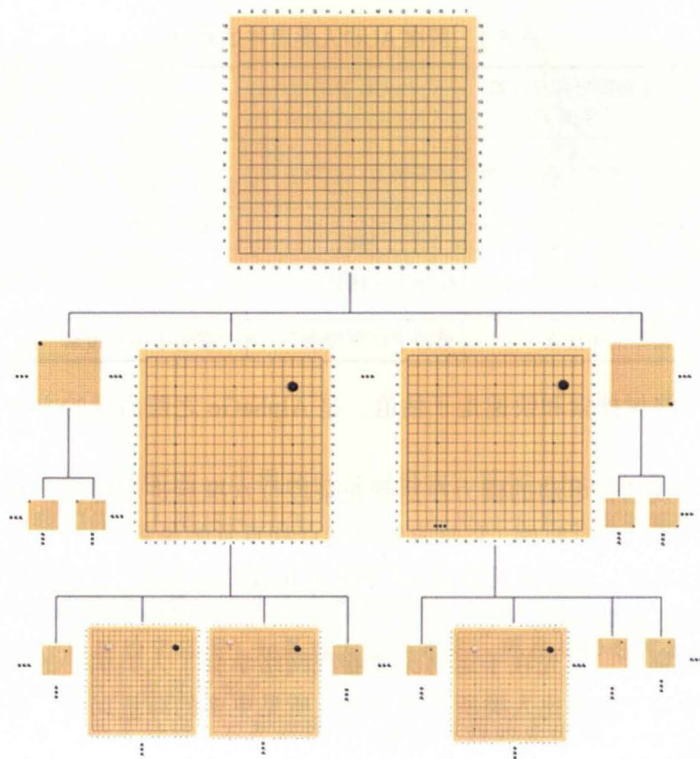


图 6-13 搜索树

根据我们的策略网络，黑棋的第 1 手，最推荐的着法位于星位（Q16），其次推荐的着法是小目（R16）。再看白棋的回应，对于黑棋的 Q16，白棋最推荐的着法是 D16，然后是 D17；对于黑棋的 R16，白棋最推荐的是 D16；等等。

这令我们想起穷举法。但这里有个难题：搜索了 2 步之后，电脑仍然不清楚局势如何。即使再走上 10 步，也不见得能看清楚。这个问题非常严重，它意味着我们仍然无法判断每一步的实际好坏。

这就是围棋的玄妙所在：很难找到一个高效的，可用程序语言描述的，可让电脑理解的，对于局势的估值函数（value function）。

对于象棋、五子棋、黑白棋等，走上 10 步，然后根据简单的估值函数，电脑就能达到相当不错的水准。但围棋的估值函数比其他棋类要难得多，如表 6-2 所示。

表 6-2 常见棋类的估值函数

	简单估值函数的核心思想
象棋	比较双方的子力情况（如，将车计为 5 分，马计为 3 分，等等）
五子棋	比较双方的棋形（如，冲四，活三，眠三，等等）
黑白棋	比较双方的自由度（即可走的位置的多少）、稳定子（即不能被吃的子），等等
围棋	地域估计（如影响力函数）、形状评估（棋形，眼位，简单的死活，等等）

造成围棋估值函数困难的原因很多。首先围棋的“死活”是个难题（盘面上有很多子，不代表这些子能生存），更重要的是还需要考察双方的发展潜力，而且各个棋块间有复杂的联系，所以不能直接比较双方的现有子力和现有地域，也不能简单地把问题拆分为对于各个棋块的评估。

早期的围棋程序，如《手谈》，需要程序作者经过反复思考和实验，人工构造“地域估计”和“形状评估”系统等，不但极其耗费时间和精力，而且棋力仍然很低，很容易遇到瓶颈。

因为如果用传统的方法（如局部特征匹配、线性回归等）构造估值函数，我们很快会发现很难向电脑描述人类的围棋概念（如“厚薄”“轻重”“子效”“味道”等）。而对于人类而言，这些概念是一个有天赋的几岁小孩就能掌握，并且一眼就能看出的。

经过多年不懈的努力，研究人员发展出 2 种自动化的估值方法，带来了围棋程序的 2 次革命：第 1 次是 MCTS 方法，第 2 次就是 AlphaGo 的价值网络。

MCTS 的威力在于，它通过合理设计搜索树的运作，即使最初只有很粗略的估值函数，随着搜索树越来越大，搜索的局面越来越多，深度越来越深，会逐渐将搜索树中所有局面的估值全面综合考虑，最终可保证得到完美的估值函数。

不过，这个向完美估值函数收敛的过程会相当漫长。如果搜索树中的估值函数存在缺陷，MCTS 就有可能出现错误判断，长期陷入陷阱，这称为“地平线效应”（horizon effect），也正是“神之一手”时所发生的事情。

简单地说，MCTS 的思想是：

- 集中搜索双方看上去最有希望的着法，减少搜索看上去不靠谱的着法。
- 但也会偶然去搜索看上去不靠谱的着法，以免漏看。
- 随着搜索的局面越来越多，它对不同着法的看法会动态变化，也会逐渐修正此前的错误看法，越来越准确。

具体可分为选择（select）、扩展（expand）、评估（evaluate）、向上传递（backup）这 4 个步骤。感兴趣的读者可参阅笔者在知乎专栏中的介绍：<https://zhuanlan.zhihu.com/p/25345778>。

6.2.4 从快速走子估值到价值网络

在 AlphaGo 出现前，MCTS 的估值方法是通过快速走子（rollout），它的方法非常粗略：既然看不清中间的局面，那么电脑不妨将棋局直接走到底，然后看最终的胜负情况。

这往往要走上几百步，因为围棋确实复杂，电脑甚至不容易理解什么叫走到底，所以电脑会走到几乎将棋盘填满，才能判断胜负（在实际围棋对局中，电脑必须学会停下来，否则会填子自杀，为此 AlphaGo Zero 为策略网络加上了额外的输出代表“停 1 手”的概率）。

快速走子估值的具体方法是，电脑会用一个走子极快（在普通电脑上，每秒就可下几万局）但棋力极弱的快速走子策略（rollout policy），把搜索树中的某个局面快速走到底 1 次，记下最后的胜负情况，称为快速走子估值（rollout value），如图 6-14 所示。

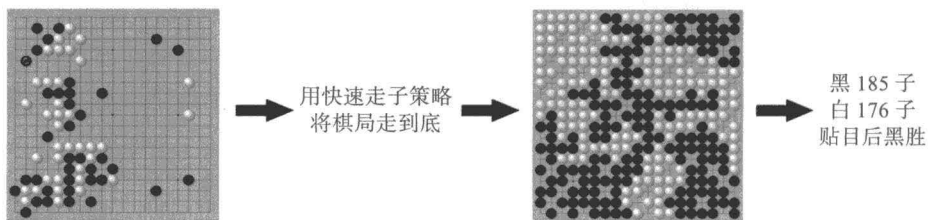


图 6-14 快速走子估值

这听上去有点像“盲人骑瞎马”，这个估值无疑会有非常大的噪音（随机性）。确实，快速走子估值很不准确，因为快速走子策略很不准确，而且对于每个局面只走 1 次（走多次会减慢速度，最终更不利于棋力）。除非当前的局面逐渐明朗，或棋局逐渐接近尾声，它才会逐渐准确。

但由于 MCTS 的设计很巧妙，因此即使用这样粗略的估值方法，再配合多种技巧，也可得到不错的结果，让电脑的棋力达到业余段位水平。

读者可能会问：我们之前已有一个棋力颇强的策略网络，为何不用它把局面走到底？这个问题很好。重要的原因是，虽然策略网络棋力远远更高，选点更准确，但运行的速度也远远更慢，会大大减少电脑搜索的局面数，最终在相同的时间内，估值的准确度反而更低。

AlphaGo 提出了比快速走子更准确的估值方法：价值网络，带来了棋力的巨大提升。价值网络可直接学会每个局面所对应的最终胜负情况。因此，如同人类棋手一样，电脑无须把棋局走到底，只需“看一眼”，就能判断局势的优劣，而且比人类棋手更为准确。

价值网络的目标是提供估值函数。在博弈论中，完美的估值函数，指的是某个局面在双方完美下完后续变化后的结果。但这个估值函数无疑很难得到。AlphaGo 的做法是用自我对弈将后续变化下完，然后用棋局的结果训练价值网络。

得到价值网络后，旧版 AlphaGo 会将价值网络与快速走子估值综合考虑（例如，取双方估值的平均值），这是因为它们之间适合互为补充：

- 有时价值网络更准确。例如对于“开放”的局面，能更早判断胜负。
- 有时快速走子估值更准确。例如在局面有“死活”这种逻辑性问题的時候。

而在 AlphaGo Zero 中，通过使用多任务学习和更深的残差网络，价值网络的精确度得以明显提高，因此不再用快速走子估值，只用价值网络估值。

最后, MCTS 会综合考虑搜索树中所有局面的估值, 进一步改进整颗博弈树的估值, 更接近完美估值函数, 如图 6-15 所示。

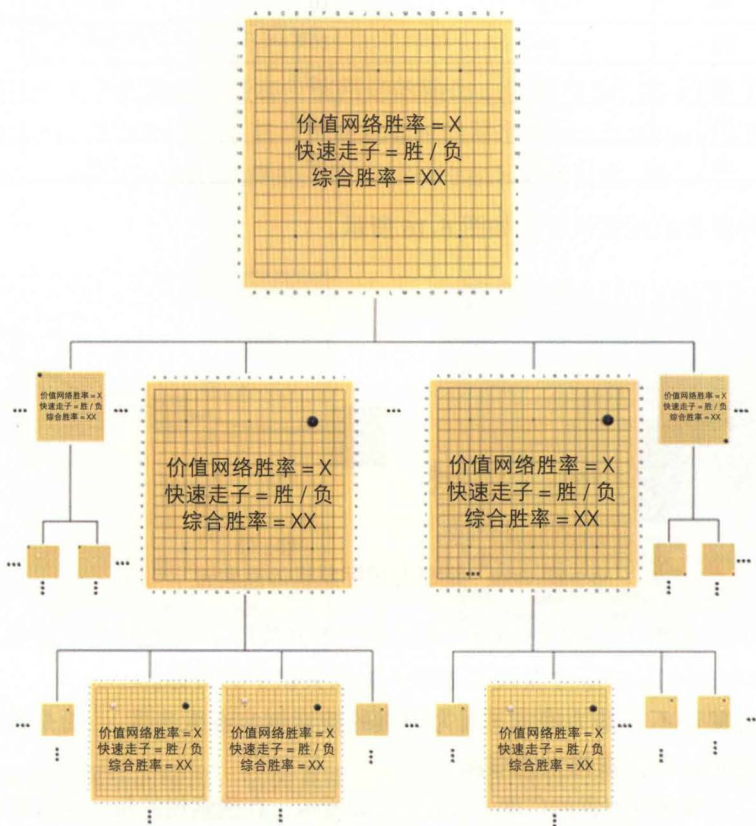


图 6-15 MCTS 的搜索树

读者可能会问: 如果 AlphaGo 的不同模块给出不一样的结果呢? 例如, 策略网络认为某一步最有可能, 而局面估值认为另一步走后的胜率最高?

MCTS 有能力综合考虑这些情况, 给出最佳答案。我们会在后文看到例子。

6.2.5 从搜索树看策略与价值网络的作用

此节从另一个角度看策略网络与价值网络的作用, 以助于读者更深入理解它们的意义。

许多初次听说电脑下棋的朋友, 会觉得电脑是在做穷举法。这种看法有一定道理, 因为电脑确实需要搜索大量的局面, 比人类所考虑的局面要多得多。但在使用策略网络和价值网络后, 电脑的搜索就会变得聪明得多, 更接近人类的思维。

从搜索树的角度看, 围棋的难点在于搜索树极广、极深, 且局面评估极难。围棋与其他常见棋类的复杂度对比, 如表 6-3 所示。

表 6-3 常见棋类的复杂度

	棋盘大小	棋盘状态数的数量级	博弈树节点数的数量级	每局平均步数	每步平均可选点数
黑白棋	64	10^{28}	10^{58}	58	10
五子棋	225	10^{105}	10^{70}	30	210
国际象棋	64	10^{47}	10^{123}	70	35
中国象棋	90	10^{40}	10^{150}	95	38
围棋	361	10^{170}	10^{360}	150	250

围棋与国际象棋的搜索树对比如图 6-16 所示。

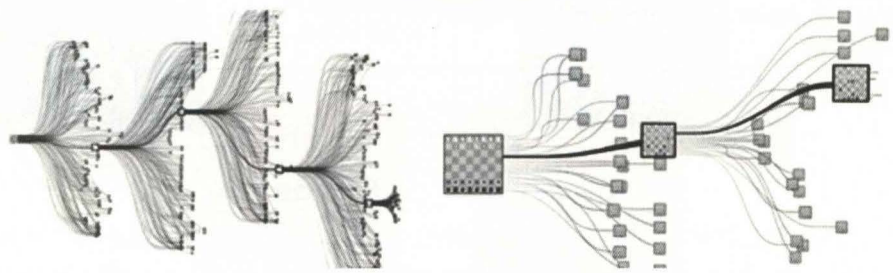


图 6-16 围棋与国际象棋的搜索树

因此，在国际象棋上 IBM 的深蓝早在 1997 年就战胜了人类大师卡斯帕罗夫，但在围棋上直到 2016 年的 AlphaGo 才实现了战胜人类顶尖棋手的目标。

AlphaGo 的秘诀不是暴力搜索（因为围棋的搜索树实在太庞大），而是使用深度卷积网络模拟人类的直觉，将搜索树大大缩减。

策略网络，负责减少搜索的广度，如图 6-17a 所示，在搜索树的每一层只需搜索少量节点。

价值网络，负责减少搜索的深度，如图 6-17b 所示，无须将棋局下到底即可得出对于局势的判断。

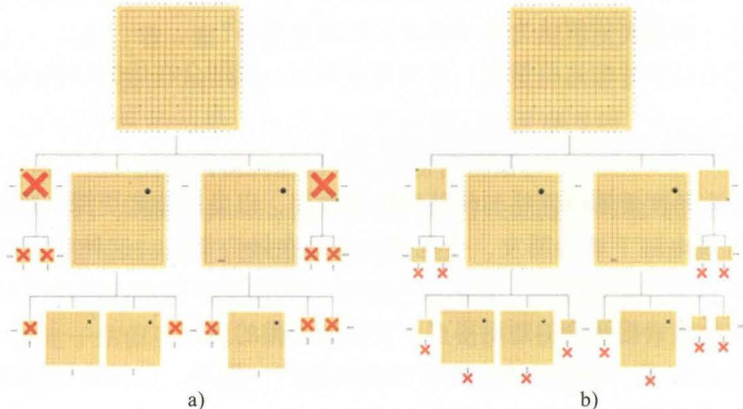


图 6-17 策略网络和价值网络对搜索的作用

在 AlphaZero 的论文中指出, 由于策略网络和价值网络非常有效, 因此 AlphaZero 在每秒搜索 4 万到 8 万步时, 就能战胜每秒搜索 3500 万到 7000 万步的未使用深度网络的对手。

6.2.6 策略与价值网络的运作实例

让我们用具体例子看策略网络和价值网络的输出, 以及与 MCTS 的结合方法。图 6-18 来自于 AlphaGo v13 在 2015 年 10 月与樊麾对弈时的某个局面, AlphaGo 持黑棋。

先提前预告, AlphaGo 最终选择的是图 6-18b 的右下角带红圈的那一手, 即价值网络认为胜率最高的那一手。

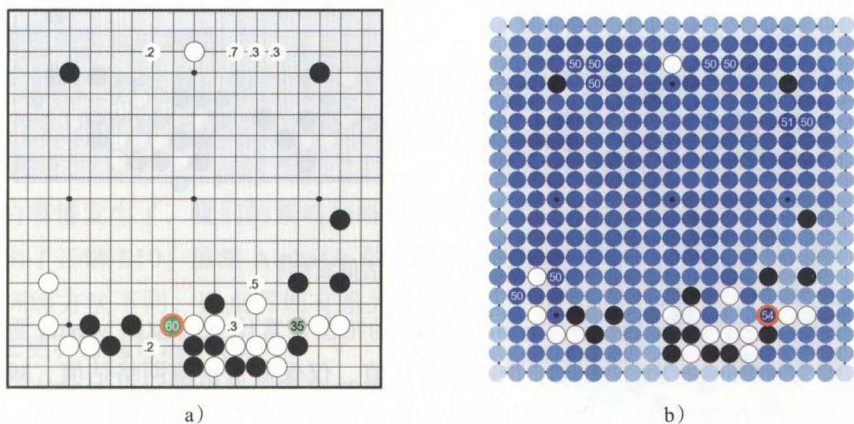


图 6-18 AlphaGo 的策略网和价值网络输出实例

图 6-18a 是策略网络的输出, 它会直接预测下一手的选点。可见, 它很好地限制了搜索的广度, 因为它对于最终着点的判断是概率 35% (注意, 这个概率不是胜率, 只是它从经验判断选择这个点的概率), 在棋盘上 300 多个可落子的着点中排名第 2 位, 很接近最终的选择。在 AlphaGo 运作时, 电脑最终选择的着点往往就在策略网络的前几位中, MCTS 会对它们优先搜索。

图 6-18b 是价值网络对下一手的直接评估。即, 电脑实验将下一手下到那个位置, 再将所生成的局面送入价值网络, 看它所输出的胜率预测。由图可见, 价值网络认为, 如果电脑的下一手在右下角带红圈的位置, 所得到的局面会具有最高的胜率, 达到 54%。这恰好就是 AlphaGo 最终的选择。

在此, 策略网络和价值网络就给出了不同的看法, 因为策略网络更推荐下方中间的一手, 概率达到 60%。MCTS 会综合考虑这些情况, 做出最终的选择。具体的过程, 请继续看图 6-19。

图 6-19a 是 MCTS 对价值网络输出胜率的改进, 即, 不但看每一手后的价值网络胜率, 还会看后续局面的价值网络胜率, 修正之前的估计。这会比最初的价值网络的胜率估计更准确。可看到, 它对于右下角那一手的评分是 53% 胜率, 仍然是最高的。

图 6-19b 类似于图 a，但不是用价值网络评估胜率，而是用快速走子评估胜率。可看到它对于那一手的评分只是 51% 胜率，而对于另一手的评分是 55% 胜率。另外我们可注意到它的胜率评估波动较大，因为快速走子估值的噪音很大。

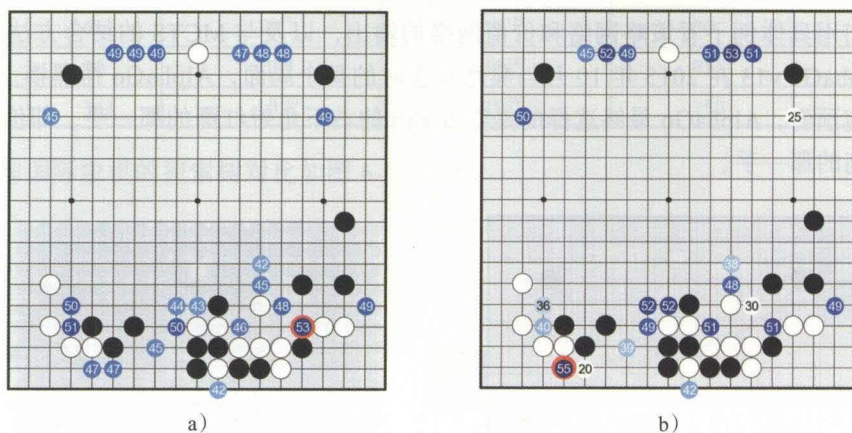


图 6-19 MCTS 的估值实例

值得一提的是，许多棋手认为 AlphaGo Zero 的招法常常比 AlphaGo Master 更自然，这很可能就是因为 AlphaGo Zero 去除了快速走子估值，仅保留了价值网络估值，减少了估值的噪音。事实上，在观看 AlphaZero 的国际象棋对局后，许多人类大师也评价 AlphaZero 的棋比此前基于海量搜索的 AI 更像人类的思维。这说明，深度网络确实与人类大脑有某种共通之处。

继续看图 6-20。

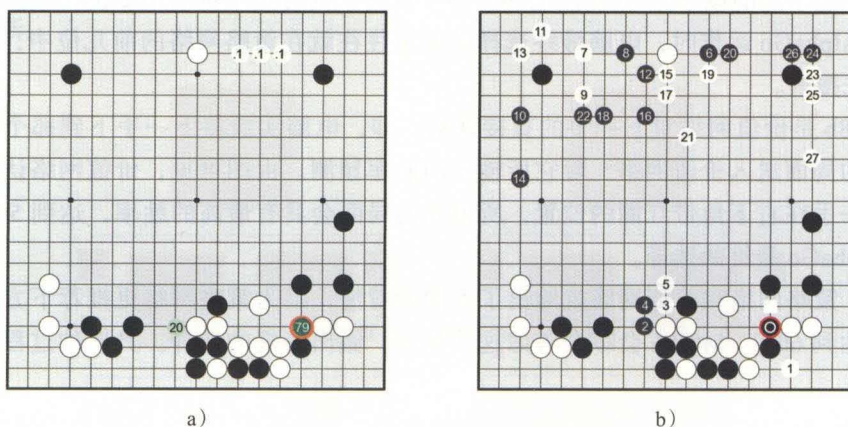


图 6-20 MCTS 的最终输出

图 6-20a 是 MCTS (结合了价值网络和快速走子策略后) 在不同着点所停留的时间比例。MCTS 的特点是, 最后选择的会是在搜索过程中停留时间最长的节点 (如前所述, MCTS 会更多地集中搜索看上去最有希望的着点)。

图中很多着点的停留时间比例很低, 是因为 MCTS 会把搜索集中在最有希望的区域。而棋盘上的许多着点是策略网络所不推荐的, 或 MCTS 经过较少的搜索就认为希望不大, 因此 MCTS 会很少对其进行深层次的搜索。注意, 根据 MCTS 的原理, 它仍然有时会去访问这些着点, 以防止看漏, 只是访问的可能性很低。

图 6-20b 是 MCTS 对于双方后续的最优着法的预测。例如图中的 1 号点, 是 AlphaGo 认为人类对于红圈那一手的最佳回应, 而樊麾当时的回应是白色方框的那一手, 赛后他表示电脑的推荐是更好的选择。

让我们回到人机大战的第 78 手, 看我们的策略网络有何见解。在此, 策略网络的目标, 是推荐第 79 手的选点, 即 AlphaGo 的回应, 如图 6-21 所示。

有趣的事情发生了, 我们的策略网络对于第 79 手竟然给出了正解 L10。虽然 AlphaGo 的策略网络与我们的训练方法不同, 选点可能有差异, 但 L10 确实看上去更符合通常的棋感。

那么, 当时 AlphaGo 出错, 很可能是由于在第 78 手后的局面错综复杂, 因此价值网络和快速走子的估值出现了问题, 从而影响了 MCTS 的结果。

如前所述, MCTS 可保证在足够长时间后收敛到最优解, 所以如果当时给 AlphaGo 更多的时间, 它最终会察觉自己的错误。但这可能会需要极其长的时间。

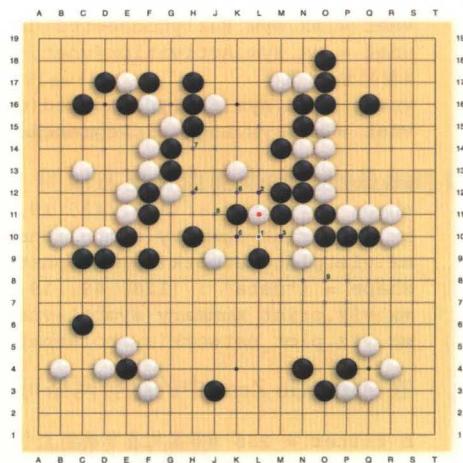


图 6-21 策略网络对“神之一手”的解读

6.3 AlphaGo 中的深度卷积网络架构

在 AlphaGo v13 和 v18 中的策略网络和价值网络可用下列 MXNet 代码定义:

```
n_filter = 192 #每层的卷积神经元个数
pnet = mx.symbol.Variable('data') #策略网络
for i in range(1, 12+1): #构建第1-12层
    if i == 1: #第1层是5*5卷积
        pnet = mx.sym.Convolution(pnet, name='conv'+str(i), kernel=(5, 5),
                                   pad=(2, 2), num_filter = n_filter)
    else: #第2-12层是3*3卷积
        pnet = mx.sym.Convolution(pnet, name='conv'+str(i), kernel=(3, 3),
                                   pad=(1, 1), num_filter = n_filter)
    pnet = mx.sym.Activation(pnet, name='act'+str(i), act_type='relu')
```



```

#合并为1个通道
pnet = mx.sym.Convolution(pnet, name='convFINAL', kernel=(1, 1), num_filter=1)
pnet = mx.sym.Flatten(pnet)
#输出为361个概率
pnet = mx.sym.SoftmaxOutput(pnet, name='softmax')

vnet = mx.symbol.Variable('data') #价值网络
for i in range(1, 13+1): #构建第1-13层
    if i == 1: #第1层是5*5卷积
        vnet = mx.sym.Convolution(vnet, name='conv'+str(i), kernel=(5, 5),
                                   pad=(2, 2), num_filter = n_filter)
    elif i < 13: #第2-12层是3*3卷积
        vnet = mx.sym.Convolution(vnet, name='conv'+str(i), kernel=(3, 3),
                                   pad=(1, 1), num_filter = n_filter)
    elif i == 13: #第13层是1*1卷积
        vnet = mx.sym.Convolution(vnet, name='conv'+str(i), kernel=(1, 1), num_
                                   filter = 1)
    vnet = mx.sym.Activation(vnet, name='act'+str(i), act_type='relu')
#全连接层
vnet = mx.sym.Flatten(vnet)
vnet = mx.sym.FullyConnected(data=vnet, name='fc1', num_hidden=256)
vnet = mx.sym.Activation(vnet, name='fc1_act', act_type='relu')
#输出为1个概率
vnet = mx.sym.FullyConnected(data=vnet, name='fc2', num_hidden=1)
vnet = mx.sym.LogisticRegressionOutput(data=vnet, name='out')

#检查参数量和结构图
shape = {"data": (128, 48, 19, 19)}
mx.viz.print_summary(symbol=pnet, shape=shape)
mx.viz.plot_network(symbol=pnet, shape=shape).view()
shape = {"data": (128, 49, 19, 19)}
mx.viz.print_summary(symbol=vnet, shape=shape)
mx.viz.plot_network(symbol=vnet, shape=shape).view()

```

在 AlphaGo Zero 中的综合网络可用下列 MXNet 代码定义：

```

n_filter = 256 #每层的卷积神经元个数
n_blocks = 39 #残差单元数
#初始卷积单元
net = mx.symbol.Variable('data')
net = mx.sym.Convolution(net, name='convPRE', kernel=(3, 3), pad=(1, 1), num_
filter = n_filter)
net = mx.sym.BatchNorm(net, name='bnPRE', fix_gamma=False)
net = mx.sym.Activation(net, name='actPRE', act_type='relu')
for i in range(1, n_blocks+1): #残差单元
    identity = net # 保存之前的输出
    net = mx.sym.Convolution(net, name='convA'+str(i), kernel=(3, 3), pad=(1, 1),
num_filter = n_filter)
    net = mx.sym.BatchNorm(net, name='bnA'+str(i), fix_gamma=False)
    net = mx.sym.Activation(net, name='actA'+str(i), act_type='relu')
    net = mx.sym.Convolution(net, name='convB'+str(i), kernel=(3, 3), pad=(1, 1),
num_filter = n_filter)
    net = mx.sym.BatchNorm(net, name='bnB'+str(i), fix_gamma=False)
    net = net + identity # 加上之前的输出，即为残差结构
    net = mx.sym.Activation(net, name='actB'+str(i), act_type='relu')

#策略输出

```



```

pnet = mx.sym.Convolution(net, name='convP', kernel=(1, 1), num_filter=2)
pnet = mx.sym.BatchNorm(pnet, name='bnP', fix_gamma=False)
pnet = mx.sym.Activation(pnet, name='actP', act_type='relu')
pnet = mx.sym.Flatten(pnet)
#输出为362个概率, 对应361个着点, 或停一手(代表棋局结束)
pnet = mx.sym.FullyConnected(data=pnet, name='fcP', num_hidden=362)
pnet = mx.sym.SoftmaxOutput(pnet, name='softmaxP')

```

#价值输出

```

vnet = mx.sym.Convolution(net, name='convV', kernel=(1, 1), num_filter=1)
vnet = mx.sym.BatchNorm(vnet, name='bnV', fix_gamma=False)
vnet = mx.sym.Activation(vnet, name='actV', act_type='relu')
vnet = mx.sym.Flatten(vnet)
#全连接层
vnet = mx.sym.FullyConnected(data=vnet, name='fc1V', num_hidden=256)
vnet = mx.sym.Activation(vnet, name='fc1_actV', act_type='relu')
vnet = mx.sym.FullyConnected(data=vnet, name='fc2V', num_hidden=1)
vnet = mx.sym.LogisticRegressionOutput(data=vnet, name='outV')

```

#检查参数数量和结构图

```

shape = {"data" : (32, 17, 19, 19)}
mx.viz.print_summary(symbol=pnet, shape=shape)
mx.viz.plot_network(symbol=pnet, shape=shape).view()
mx.viz.print_summary(symbol=vnet, shape=shape)
mx.viz.plot_network(symbol=vnet, shape=shape).view()

```

这里和 AlphaGo 论文中的定义有一些细微区别, 例如这里价值网络的输出是 0 到 1, 而 AlphaGo 论文中的输出是 -1 到 1。

6.4 AlphaGo 的训练过程

6.4.1 原版 AlphaGo : 策略梯度方法

在原版 AlphaGo 中, 网络的训练可分为 3 个步骤, 如图 6-22 所示。

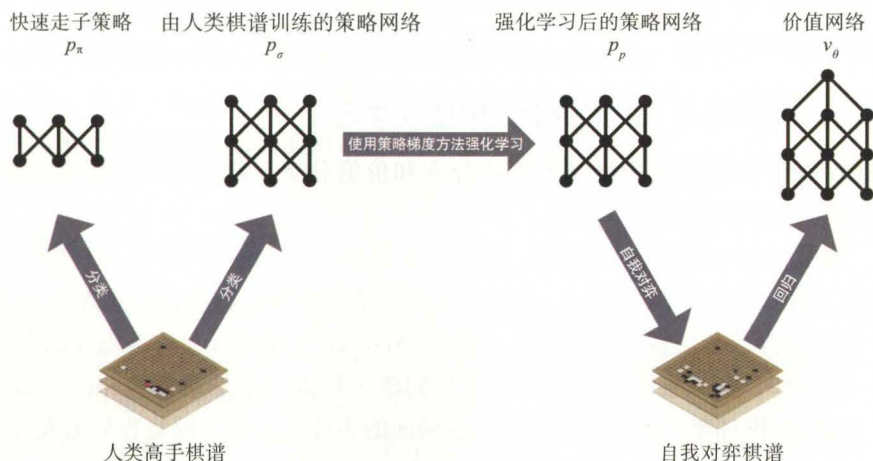


图 6-22 原版 AlphaGo 的训练过程

1) 初步训练。通过学习人类高手棋谱, 获得策略网络。这个过程相对简单, 需要的计算资源较少, 我们会在本书中实战训练。

2) 强化训练。通过强化学习中的策略梯度 (policy gradient) 方法, 增强策略网络。首先, 按照策略网络给出的选点概率, 进行自我对弈。然后:

- 对于最终赢棋的一方, 说明这次选择的策略可能是正确的。所以对于这次所经历的每一个局面, 都加强选择这局的走法的概率。
- 对于最终输棋的一方, 说明这次选择的策略可能是错误的。所以对于这次所经历的每一个局面, 都减少选择这局的走法的概率。

例如, 如果黑棋开局走在“星位”, 白棋回应走在“小目”, 最后白棋输了, 那么黑棋会加强开局走星位的概率, 以及后续每一步选择这局的走法的概率; 白棋会减少在黑棋开局走星位的情况下走小目的概率, 以及后续每一步选择这局的走法的概率。

可以想象, 这个过程的噪声很大。因为白棋可能并不是败在开局, 而是败在中盘的某一步; 黑棋也许并不是真得走对了策略, 而是因为白棋看漏了一步, 才因此侥幸取胜。

不过, 在大量对局后, 噪声会逐渐减少。这其实就像人类棋手找到适合自己棋风的过程。

以上是最简单的策略梯度的例子。它的一个问题是可能陷入局部最优, 因为对付自己有效, 不代表对付其他对手有效。因此 AlphaGo 会建立一个对手池, 包括整个进化过程中形成的所有策略, 保证新策略尽量对于不同对手都有效。

在此基础上可做各种改进, 例如配合价值网络, 更精确地看到败着在哪里, 经典的方法是 REINFORCE 算法: 如果价值网络判断当前的形势已经很差, 那么后续的着法就不足以称为败着; 如果输掉了这盘棋, 那么败着一定是在看上去形势好像不错的时候走出来的。

3) 训练价值网络。这个过程较为直接, 使用自我对弈生成的棋谱局面和最终结果即可训练, 难点在于需生成大量自我对弈棋谱, 对运算资源的需求很高。

为防止过拟合, 原版 AlphaGo 只在每个自我对弈棋谱中选择 1 个局面用于训练价值网络。新版 AlphaGo 通过合并策略网络和价值网络, 使网络更不容易出现过拟合, 可使用棋谱中的所有局面进行训练。

6.4.2 新版 AlphaGo : 从蒙特卡洛树搜索学习

AlphaGo Zero 的深度卷积网络的策略分支和价值分支, 都是完全从自我对弈中学习, 具体说来, 是从 MCTS 过程中学习。

首先, 使用 MCTS 进行自我对弈, 如图 6-23 所示。

- 对于对弈中的每个局面 $\{s_i\}$, MCTS 在不同落子位置 a 的停留时间 $N(s_i, a)$ 不同。
- 令最终选择落子位置 a 的概率 π_a 正比于 $N(s_i, a)^{1/\tau}$, 其中 τ 是称为温度的参数。
 - 若 $\tau \rightarrow 0$, 则只会选择停留时间最长的落子位置, 这是通常的 MCTS 做法。如果这样自我对弈, 棋力最强, 但生成局面的多样性不够, 网络容易出现盲点。
 - 若 $\tau \rightarrow \infty$, 则会完全随机选择落子位置, 局面变化最多, 棋力最差。

- 因此，选取一个适当的 τ ，可在棋力与局面多样性之间找到平衡，有利于训练。
- 令棋局的最终结果为 z 。将 z 和每个局面 s_i 所对应的 π_i 存储。

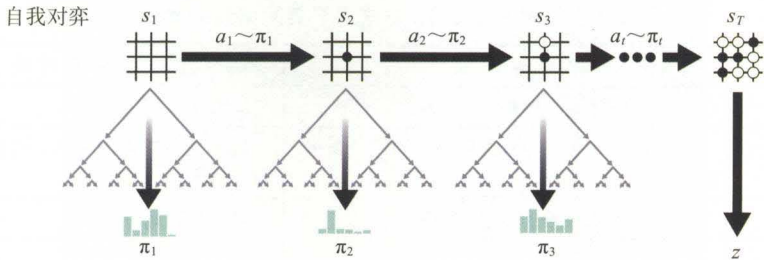


图 6-23 AlphaGo Zero 的自我对弈

在得到一批自我对弈对局后，将它们用于训练网络 f 的策略输出和价值输出。对于每个局面 s_i ，策略输出 p 应尽量等于 π_i ，价值输出 v 应尽量等于最终的结果 z ，如图 6-24 所示。

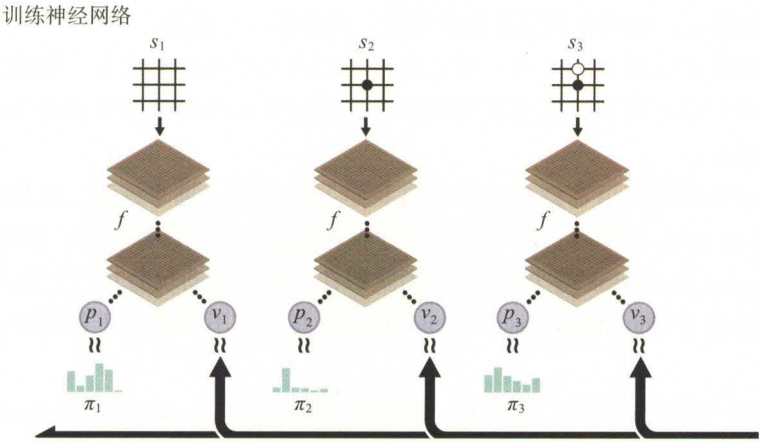


图 6-24 AlphaGo Zero 的训练过程

训练后得到更强的网络，可生成更高质量的自我对弈棋谱。重复这一过程，电脑的棋力就越越来越强。而且，AlphaGo Zero 的方法在此比传统的强化学习方法更为稳定可靠，因为 MCTS 可保证收敛到完美的估值函数。

6.5 AlphaGo 方法的推广

AlphaGo 属于深度学习和强化学习的结合，可称为深度强化学习。这是一个极具威力的组合，因为深度学习的容纳能力（capacity）很强，形象地说就是大脑容量很大，有能力学会在自我对弈中遇到的各种复杂现象，不断进步。

这已接近科幻小说中的场景：AI 在拥有了自我提升的方法后，成长的速度惊人，不久就达到了“神”一般的境界。那么，AlphaGo 的方法是否可用于解决更多现实生活中的问题，

而不仅仅是围棋或棋类游戏？

确实，我们可如表 6-4 所示进行对比。

表 6-4 将 AlphaGo 方法用于实际问题的尝试

问题描述	策略网络	价值网络
下棋	下一步应该走在哪里	最大化胜率
自动驾驶	下一步应该如何控制方向盘、油门、刹车、换挡等	最小化与终点的距离，同时最大化安全度和舒适度
炒股	下一步应该买入 / 卖出 / 等待	最大化利润
...

但是围棋与炒股有重要的区别。对于围棋而言，存在可从理论上保证得到完美估值函数的方法：MCTS。这是可用数学精确定义的目标。因此，随着 AlphaGo 的不断进化，它就能越来越接近这个完美的目标。

而对于股市而言，如何找到完美估值函数，无疑是一个难题。

自动驾驶的研究人员提出了另一种思路：我们先模拟现实世界，把现实世界变为一种游戏。虽然我们很难模拟一座真实的城市，但我们可先从简单的驾驶游戏做起，模拟一座简单的游戏城市。

于是，自动驾驶程序可在游戏世界中自我学习和探索，越来越强。这种环境也很适合于端对端（end-to-end）的学习，即程序直接分析观察到的游戏屏幕画面，做出下一步决策。

因此，Google、Facebook 等公司的 AI 团队都在“电脑自动玩游戏”上投入了许多研究。对此感兴趣的读者可访问 <https://gym.openai.com>，其中有不少适用于机器学习的简单模拟环境。

通过游戏还可快速生成大量数据。例如在 http://download.visinf.tu-darmstadt.de/data/from_games/ 和 <http://www.europe.naverlabs.com/Research/Computer-Vision/Proxy-Virtual-Worlds> 中，研究人员通过游戏自动生成大量街景图和对应的精确标记，如图 6-25 所示。

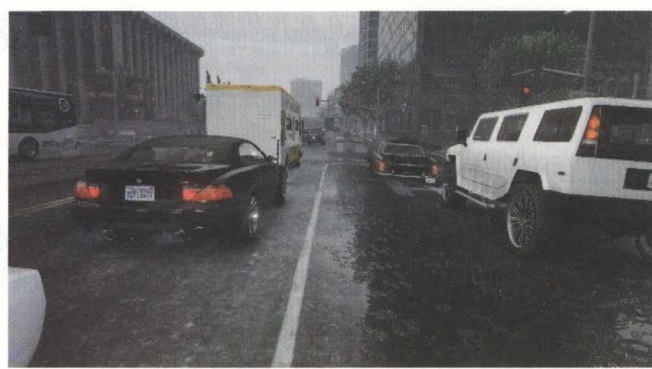


图 6-25 由游戏生成的训练数据



图 6-25 (续)

如果深度网络可由这种数据准确识别出图像中每个点所对应的物体类型，对于自动驾驶等领域将很有裨益。

训练策略网络与实战

根据 CGI 和 Leela 等知名围棋 AI 作者的估算，如果用 AlphaGo Zero 的方法从零开始训练，需要相当于 10 000 张 GTX 1080 Ti 的算力用于自我对弈。这对于绝大多数读者不现实，因此本章的目标是从人类高手棋谱训练策略网络。

目前 Leela 的作者已发起了开源的众包训练计划 (<http://zero.sjeng.org>) 并已取得相当好的进展，感兴趣的读者可参与训练出与 AlphaGo Zero 相近的 Leela Zero。

7.1 训练前的准备工作

在此简要说明如何生成策略网络的训练数据。

为方便读者实验，笔者提供了已制作好的小型训练数据包，读者可在 <https://github.com/BlinkDL> 下载。其中包括奕城 (Tygem) 9 段在 2004 到 2005 年的 20 000 多局对局 (来自 <http://baduk.sourceforge.net/TygemAmateur.7z>)。

具体格式如下：

- ❑ 分成 598 个数据包，默认 0 到 5 号数据包用于测试，6 到 597 号数据包用于训练。
- ❑ 每 363 个字节对应一个局面下的落子位置和特征层情况。前 2 个字节代表落子位置的 X 和 Y 坐标，后 361 个字节代表本书第 4 章所述的 8 个特征层 (每个字节的二进制 8 位分别代表 8 个特征层的值)。
- 特征层的排列顺序是：最后一手的位置，有 4 气及以上的棋子，只有 3 气的棋子，只有 2 气的棋子，只有 1 气的棋子，是否是空点，是否是对方棋子，是否是本方棋子。

7.1.1 棋谱数据

首先需要棋谱数据。最常见的棋谱格式是 SGF，其定义见 <http://www.red-bean.com/sgf/>。

国外著名的围棋服务器是 KGS，它的对局可以在 <https://www.u-go.net/gamerecords/> 下载，即为 SGF 格式。

此外可在 <https://github.com/yenw/computer-go-dataset> 获取奕城（Tygem）服务器的棋谱，CGOS（电脑围棋服务器）上的 AI 对战棋谱，以及职业围棋手的对弈棋谱，等等。

以 KGS 的一盘棋谱为例，解释 SGF 格式：

```
(;GM[1] 这是棋的类别，1代表围棋
FF[4] 这是SGF格式的版本号，4是最新版
SZ[19] 这是棋盘大小，这里是19路
PW[guest0] 这是持白棋手的名称，这里是guest0
WR[6d] 这是持白棋手的段位，这里是KGS 6段
PB[twoeye] 这是持黑棋手的名称，这里是twoeye
BR[6d] 这是持黑棋手的段位，这里是KGS 6段
DT[2017-05-01] 这是棋谱的日期
PC[The KGS Go Server at http://www.gokgs.com/] 这是棋局所在的场所
KM[6.50] 这是棋局的贴目，这里是6.5目
RE[B+Resign] 这是棋局的结果，B+代表黑胜，W+代表白胜。Resign代表是对方认输
RU[Japanese] 这是棋局的规则，这里是日本规则
CA[UTF-8] 这是棋局的字符编码
ST[2] 这与棋谱的显示方式有关，可以忽略
AP[CGoban:3] 这是生成棋谱的程序名
TM[0] 这是棋局的限时情况，这里代表无限时
OT[80x10 byo-yomi] 这是棋局的读秒规则
B[pp];W[dd];B[pc];W[dq];B[do];W[co];B[cn];W[cp];B[cf];W[di];B[ef];W[fd];B[bd];W
[cc];B[bi];W[qe];B[od];W[ic];B[qh];W[nq];B[pn];W[pr];B[qq];W[kq];B[dn];W[ep];B[d
p];W[fq];B[cq];W[bq];B[cr];W[br];B[bn];W[bp];B[gm];W[gi];B[pf];W[dg];B[df];W[bh
];B[ch];W[cil];B[bj];W[bj];B[ah];W[dk];B[bk];W[aj];B[ck];W[cj];B[dl];W[ek];B[dh];W
[eh];B[eg];W[el];B[em];W[cl];B[bl];W[dm];B[fn];W[eol];B[cm];W[dl];B[en];W[hl];B[g
p];W[hm];B[gq];W[er];B[hn];W[in];B[gl];W[ho];B[gk];W[fh];B[gn];W[ij];B[io];W[ip
];B[jo];W[jn];B[ko];W[hp];B[fj];W[ei];B[lq];W[lr];B[jq];W[jr];B[kp];W[kr];B[iq];W
[kn];B[ln];W[lm];B[mn];W[mm];B[nm];W[nl];B[om];W[lo];B[jp];W[mo];B[nn];W[lp];B[i
r];W[mk];B[pk];W[lc];B[gf];W[ih];B[ob];W[oi];B[pi];W[oj];B[ok];W[nk];B[oh];W[nh
];B[ng];W[og];B[ph];W[mg];B[nf];W[mf];B[me];W[le];B[md];W[ld];B[hj];W[hk];B[ik];W
[im];B[ii];W[gj];B[jj];W[hi];B[jh];W[ig];B[mh];W[ni];B[lh];W[kg];B[lk];W[mj];B[k
h];W[ij];B[jg];W[kf];B[jf];W[if];B[je];W[ie];B[hj];W[lj];B[jl];W[ij];B[jc];W[jb
];B[hj];W[km];B[l1];W[ij];B[jd];W[kb];B[hj];W[kl];B[kk];W[ij];B[id];W[ji];B[hc];
W[hd];B[ib];W[gc];B[gb];W[fb];B[lf];W[lg];B[ke];W[lf];B[mc];W[mb];B[nb];W[ga];B
[ia];W[hb];B[ic];W[qr];B[ma];W[la];B[lb];W[ai];B[bc];W[mb];B[na];W[bh];B[cg];W[
bb];B[cd];W[kc];B[ja];W[db];B[ki];W[]) 这是棋谱，B代表黑方，W代表白方
```

在棋谱中：

- [] 内为落子坐标，a 代表 1，s 代表 19。
- 横坐标为从左到右，纵坐标为从下到上。
- 注意，棋谱图像的横坐标会把字母 I 跳过，因为它可能会与数字 1 混淆。但 SGF 棋谱不会跳掉。举例，[pp] 代表右上角的星位。
- 没有坐标的 []，就代表停一手（pass），这通常意味着棋局结束。

7.1.2 落子模拟

由于棋谱中只有双方的落子位置，还需用程序生成每步后的棋盘局面，以及对应的特征层。

这个过程容易出现 bug，建议读者用棋谱检验，如果在模拟棋谱的落子过程中，出现落在已有棋子的地方，或落在禁入点上，就说明程序存在 bug，或棋谱本身有错误。

如图 7-1 所示，对于左边的情况，被围起来的空点是黑棋的禁入点，因为黑棋走进去是自杀。对于右边的情况，在打劫规则允许时，被围起来的空点不是黑棋的禁入点，因为黑棋走进去可先提掉白棋。

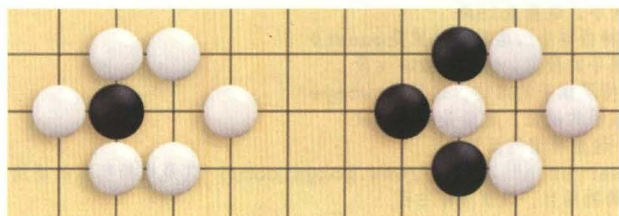


图 7-1 围棋中的禁入点

落子过程的逻辑如下：

- 1) 是否在棋盘内？是否为空点？是否为打劫的禁入点？
- 2) 是否可吃掉对方的子？如果是，就将它们吃掉。
- 3) 如果不能吃子，且会自杀，则也属于禁入点。
- 4) 更新棋盘，更新劫的情况。
- 5) 严格说，还应处理“同形再现”，不过它的概率很低，可暂先忽略。

具体代码可参考 <https://github.com/pasky/michi> 和 <http://www.yss-aya.com/20151114dentsu.zip>。

7.1.3 终局判断

对于围棋的初学者和电脑，甚至不容易判断一盘棋在什么时候是下完了。简单的想法是不能“填自己的眼”，但围棋中还有许多微妙的局面，如“双活”，双方都会避免在这个区域下子，然而双活很难用简单的方法判断。

例如，图 7-2 中用圆圈标记的黑棋和白棋属于双活，它们共享 a 和 b 这两口气。双方都不会走在 a 或 b，因为走了就会被对方吃掉。

一盘棋下完有两种情况，一种是一方主动认输，另一种是双方都不愿意再下，都选择停一手 (pass)。

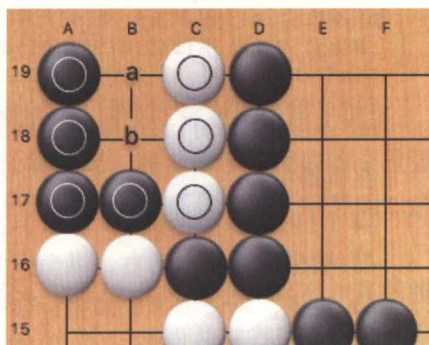


图 7-2 围棋中的双活

我们可将停一手也作为可能的选项，送入策略网络学习；或用 MCTS 解决：如果发现胜率过低，就应该认输；如果发现停一手是此时的最佳选择，就应该停一手。

另一个问题是如何最后判断胜负。在电脑围棋中通行的规则是 Tromp-Taylor 规则，与中国规则相似，介绍见：<http://tromp.github.io/go.html>。

在实际与人类的对局中，还需要在终盘移除死子，这可通过 MCTS 判断：如果在搜索树中发现某些棋子很难生存，就可认为它们是死子。

7.2 训练代码

训练策略网络所需的数据量实际很大。如果使用 10 万局棋谱，平均每局有 200 步，那么总局面数就高达 2000 万个。这里有几点需注意：

- ❑ 推荐使用多线程载入数据，以保持 GPU 全力工作，毋需频繁等待 CPU 供给数据。
- ❑ 数据增强很重要。需将棋盘局面按 8 种对称性随机变换（原始局面，旋转 90 度，旋转 180 度，旋转 270 度，镜像，旋转 90 度并镜像，旋转 180 度并镜像，旋转 270 度并镜像），这可大大减少过拟合。

我们在此的训练程序分为 4 个文件：

- ❑ train.py：这是主程序。
- ❑ config.py：这里储存训练用的参数。
- ❑ util.py：这里储存一些辅助函数。
- ❑ symmetry.py：负责将棋盘局面按 8 种对称性随机变换。

7.2.1 主程序：train.py

首先，初始化库，并设置部分常量。

```
# -*- coding: utf-8 -*-
import numpy as np
import os
import gc
import time
import struct
import sys
import multiprocessing
import random
import mxnet as mx
from mxnet.ndarray import concatenate
from mxnet.io import DataIter, DataDesc
from collections import OrderedDict, namedtuple

# 载入自定义的辅助库
import config
import symmetry
from util import MyNetwork, my_print
```



```
# 定义需监控的性能指标
my_metric = [mx.metric.create('acc'), # 准确率
              mx.metric.create('ce'), # 交叉熵损失
              mx.metric.create('top_k_accuracy', top_k = 2), # 前2位命中率
              mx.metric.create('top_k_accuracy', top_k = 5), # 前5位命中率
              mx.metric.create('top_k_accuracy', top_k = 10)] # 前10位命中率

# 储存部分指标
epoch_accuracy = 0
epoch_loss = 0
epoch_loss_last = -1
time_last = None # 监控训练耗时
```

定义主函数，定义网络。

```
# 主函数
def main(argv):

    # 定义网络，这里使用较小的网络，有训练资源的读者可增大它的层数和神经元数
    # 这里的MyBlocks函数的定义在随后的util.py中
    net = mx.sym.Variable('data')
    net = MyNetwork().MyBlocks(data=net, blocks=1, block_type="C", name="pre", n_
        filter=128, kernel=3)
    net = MyNetwork().MyBlocks(data=net, blocks=6, block_type="R-BACBAC",
        name="", n_filter=128, kernel=3)
    net = MyNetwork().MyBlocks(data=net, blocks=1, block_type="BAC",
        name="final", n_filter=1, kernel=1)
    net = mx.sym.Flatten(data=net)
    net = mx.sym.SoftmaxOutput(data=net, name='softmax')
    # 输出参数情况
    shape = {"data" : (1, 8, 19, 19)}
    mx.viz.print_summary(symbol=net, shape=shape)
```

处理模型的载入（然后可随时中断训练，继续训练）。

```
# 建立输出模型目录
if not os.path.exists(config.model_directory):
    os.mkdir(config.model_directory)
# 建立日志文件
log_file = os.open(config.model_directory + "_train.csv", os.O_RDWR|os.O_
    CREAT|os.O_APPEND)

# 新建模型，或载入此前的模型
if config.n_epoch_load == 0:
    module = mx.mod.Module(symbol=net, context=config.train_device)
    arg_params = None
    aux_params = None
else:
    sym, arg_params, aux_params = mx.model.load_checkpoint(config.model_prefix,
        config.n_epoch_load)
    module = mx.mod.Module(symbol=sym, context=config.train_device)
```

准备训练，包括建立迭代器、监控性能指标。

```
# 建立迭代器，这里最后一个参数代表是否用于训练（训练时需进行数据增强，测试时不需要）
data_iter = MyDataIter(config.batch_size, True)
```

```

val_iter = MyDataIter(config.batch_size, False)

# 如前所述, 我们将棋谱数据分为多个文件, 这里定义每完成1个文件为1个虚拟epoch
def epoch_callback(epoch, symbol, arg_params, aux_params):
    global time_last, epoch_accuracy, epoch_loss, epoch_loss_last

    # 输出真正的epoch数 (每完成1次全部文件的训练, 为一个真正的epoch)
    real_epoch = float(epoch) / (config.train_end_index - config.train_begin_index + 1)
    my_print(' %.2f', (real_epoch))

    # 输出性能指标
    batch_accuracy = my_metric[0].get_name_value()[0][1]
    cross_loss = my_metric[1].get_name_value()[0][1];
    my_print(' : 1/2/5/10 %.2f', (100.0 * batch_accuracy))
    my_print('-%.2f', (100.0 * my_metric[2].get_name_value()[0][1]))
    my_print('-%.2f', (100.0 * my_metric[3].get_name_value()[0][1]))
    my_print('-%.2f%%', (100.0 * my_metric[4].get_name_value()[0][1]))
    my_print(' ce %.3f', (cross_loss))
    # 输出当前学习速率
    my_print(' : lr %.4f' % (config.learning_rate))

    # 更新一些性能指标
    epoch_accuracy += batch_accuracy;
    epoch_loss += cross_loss;

    # 输出训练耗时
    time_now = time.time()
    if time_last is None:
        time_last = time_now
        my_print(' : n/a\n')
    else:
        my_print(' : %.2fs\n' % (time_now - time_last))
    time_last = time_now

```

定期保存模型, 并测试性能, 以此调整学习速率。

```

# 定期保存模型, 并测试性能, 以此调整学习速率
if epoch % config.save_period == 0:
    # 保存模型
    module.save_checkpoint(config.model_prefix, epoch, save_optimizer_
        states=True)

    # 测试在测试数据集的性能指标
    val_metric = module.score(val_iter, [mx.metric.Accuracy(), mx.metric.
        CrossEntropy()])
    val_accuracy = val_metric[0][1]
    val_loss = val_metric[1][1]
    # 输出性能指标
    epoch_accuracy = float(epoch_accuracy) / config.save_period
    epoch_loss = float(epoch_loss) / config.save_period
    print("=== [saved] : train %.2f%% ce %.3f : validate %.2f%% ce %.3f ===" %
        (100.0 * epoch_accuracy,
         epoch_loss,
         100.0 * val_accuracy,
         val_loss))
    # 保存到日志中

```

```

os.write(log_file, str(real_epoch) + "," + str(epoch_accuracy) + ","
        + str(val_accuracy) + "," + str(config.learning_rate) + "\n")
os.fsync(log_file)

# 自适应地减少学习速率：如果在训练数据的平均损失提高了，则减少学习速率
if epoch_loss > epoch_loss_last and epoch_loss_last != -1:
    config.learning_rate = config.learning_rate * config.learning_decay
epoch_loss_last = epoch_loss
epoch_accuracy = 0.0
epoch_loss = 0.0
# 学习速率很低时，终止训练
if config.learning_rate < config.exit_learning_rate:
    exit(0)

```

开始训练：

```

def batch_callback(epoch):
    data_iter.can_load_file.set() # 训练开始，发送信号表示CPU可接着加载后续的文件

# 开始训练
module.fit(
    data_iter,
    eval_data = None, # 我们会自行测试
    eval_metric = my_metric,
    initializer = mx.initializer.MSRAPrelu(factor_type='avg', slope=0.0),

    optimizer = 'sgd',
    optimizer_params = {'learning_rate': config.learning_rate, 'wd': config.
                        wd, 'momentum': 0},

    num_epoch = 9999999, # 我们会自行决定何时终止训练
    batch_end_callback = batch_callback,
    epoch_end_callback = epoch_callback,
    arg_params = arg_params,
    aux_params = aux_params,
    begin_epoch = config.n_epoch_load + 1 # 延续之前的训练进度
)

```

这里使用自定义的数据迭代器，它会建立数据队列，并在一个新线程载入数据。

与我们自定义的数据迭代器有关

```

class OneBatch(object):
    def __init__(self, data, label, pad=None, index=None,
                 bucket_key=None, provide_data=None, provide_label=None):
        self.data = data
        self.label = label
        self.pad = pad
        self.index = index
        self.bucket_key = bucket_key
        self.provide_data = provide_data
        self.provide_label = provide_label

```

我们自定义的数据迭代器

```

class MyDataIter(DataIter):

```

```

    # 负责载入文件，这个函数将在新线程中运行

```



```

def load_file(self):
    while True: # 是一个死循环，但会等待信号，因此不会卡死
        filename = ""

        # 获取需载入的文件名
        if self.is_train:
            index = self.train_list[self.train_index]
            my_print('[pl %s' % str(index).rjust(4))
            filename = config.train_prefix + '-' + str(index) + '.dat'
        else:
            filename = config.train_prefix + "-val.dat"
            my_print('[pl validate data')

        # 载入数据文件
        with open(filename, 'rb') as data_file:
            data = bytearray(data_file.read())

        # 加入随机变换
        if self.is_train and config.apply_symmetry:
            symmetry.apply_random_symmetry(data)

        # 将读入的文件设置为正确的数组格式
        data = np.array(data).reshape(-1, config.head+361)

        # 打散数据
        if self.is_train and config.shuffle_data:
            np.random.shuffle(data)

        # 将落子位置由XY坐标变为0-360的数
        label = np.array(map(lambda x, y : x + y * 19, \
                             data[:, 0:1].flatten(), data[:, 1:2].flatten()), \
                          np.uint16)

        # 获取特征层，设置为正确的数组格式
        data = np.unpackbits(data[:, config.head:config.head+361]).reshape(-1,
                                     19, 19, 8)
        data = np.moveaxis(data, -1, 1)

        # 载入完成
        my_print(']')

        if self.is_train: # 如果是载入训练数据，则需要不断载入
            # 将数据填充到队列中
            self.queue.put(obj = [data, label], block = True, timeout = None) # save
            self.train_index = self.train_index + 1
            # 如已完成全部文件的训练，则重新打散文件的顺序
            if (self.train_index >= len(self.train_list)):
                self.train_index = 0
                random.shuffle(self.train_list)
            else: # 如果是载入测试数据，则可一次载入
                self.data_list = [mx.ndarray.array(data, config.data_device),
                                  mx.ndarray.array(label, config.data_device)]

        gc.collect() # 要求垃圾回收，否则会耗费大量内存

        if not self.is_train:

```

```

        return # 如果是载入测试数据，则一次就已完成载入

    # 否则会停下来等待信号
    if self.is_train:
        self.can_load_file.wait()
        self.can_load_file.clear()

```

数据迭代器的代码较长，可参考 MXNet 源码中的数据迭代器例子。

```

# 负责加载数据
def init_data(self):
    if self.is_train:
        tmp = self.queue.get(block = True, timeout = None) # 从队列加载数据
        self.data_list = [mx.ndarray.array(tmp[0], config.data_device),
                           mx.ndarray.array(tmp[1], config.data_device)]

    # 按MXNet所要求的规范设置
    self.data = [['data', self.data_list[0]]]
    self.label = [['softmax_label', self.data_list[1]]]
    # 设置数据个数
    self.num_data = self.data_list[0].shape[0]
    assert self.num_data >= self.batch_size, "batch_size need to be smaller
        than data size."

# 负责初始化迭代器
def __init__(self, batch_size=1, is_train = True):
    super(MyDataIter, self).__init__()

    self.can_load_file = multiprocessing.Event()

    self.cursor = -batch_size
    self.batch_size = batch_size

    # 打散加载文件的列表
    self.train_index = 0
    self.train_list = range(config.train_begin_index, config.train_end_index+1)
    random.shuffle(self.train_list)

    self.is_train = is_train
    if self.is_train: # 如果是训练数据，则开启队列和加载数据的线程
        if __name__ == '__main__':
            self.queue = multiprocessing.Queue(maxsize = 1)
            self.worker = multiprocessing.Process(target = self.load_file)
            self.worker.daemon = True
            self.worker.start()
            self.init_data()
            self.init_misc()
        else: # 如果是测试数据，则直接加载数据
            self.load_file()
            self.init_data()
            self.init_misc()

```

下面是一些细节函数，基本来自于 MXNet 源代码中迭代器的定义，无须特别关注。

```

def init_misc(self):
    self.num_source = len(self.data_list)

```

```

self.provide_data = [ DataDesc(k, tuple([self.batch_size] + list(v.
    shape[1:])), v.dtype)
    for k, v in self.data ]
self.provide_label = [ DataDesc(k, tuple([self.batch_size] + list(v.
    shape[1:])), v.dtype)
    for k, v in self.label ]

def hard_reset(self):
    self.cursor = -self.batch_size

def reset(self):
    if self.is_train:
        self.init_data()
    self.cursor = -self.batch_size

def next(self):
    self.cursor += self.batch_size
    if self.cursor < self.num_data:
        return OneBatch(data=self.getdata(), label=self.getlabel(), pad=self.
            getpad(), index=None)
    else:
        raise StopIteration

def _getdata(self, data_source):
    if self.cursor + self.batch_size <= self.num_data: # no pad
        return [x[1][self.cursor:self.cursor+self.batch_size] for x in data_source]
    else: # with pad
        pad = self.batch_size - self.num_data + self.cursor
        return [concatenate([x[1][self.cursor:], x[1][:pad]]) for x in data_source]

def getdata(self):
    return self._getdata(self.data)

def getlabel(self):
    return self._getdata(self.label)

def getpad(self):
    if self.cursor + self.batch_size > self.num_data:
        return self.cursor + self.batch_size - self.num_data
    else:
        return 0

```

最后，运行主函数。

```

# 运行主函数
if __name__ == '__main__':
    main(sys.argv)

```

7.2.2 训练参数：config.py

在此是训练中的参数和超参数，可根据情况设定。

```

#-*-coding:utf-8-*-
import numpy as np
import mxnet as mx

```



```

np.core.arrayprint._line_width = 120 # numpy输出行宽

data_device = mx.gpu()
train_device = [mx.gpu()]
auto_tune = 'fastest' # 使用最快的卷积算法

model_directory = "model" # 储存模型的目录
model_prefix = model_directory + "//model" # 模型的文件名前缀
n_epoch_load = 0 # 从哪个epoch继续训练

apply_symmetry = True # 是否使用棋盘的8种对称性
shuffle_data = True # 是否打散数据

save_period = 60 # 储存模型+测试模型+调整学习速率的间隔

batch_size = 256 # 批大小
learning_rate = 0.1 # 学习速率
wd = 0 # L2正则化强度
learning_decay = 0.9 # 每次调整学习速率的衰减度
exit_learning_rate = 0.01 # 在学习速率衰减到多少时退出训练

train_prefix = "tygem//TygemAmateur" # 数据的文件名前缀
train_begin_index = 6 # 训练开始于哪个文件
train_end_index = 597 # 训练结束于哪个文件

head = 2 # 数据文件中每个局面的头部的大小（这里表示2个字节储存落子位置）

```

7.2.3 辅助函数：util.py

这里是训练中的辅助函数，以及辅助定义网络架构的函数。

```

#-*-coding:utf-8-*-
import os
import sys
import numpy as np
import mxnet as mx
import config

# 保证输出的字符可立即显示
def my_print(str, *args):
    sys.stdout.write(str % args)
    sys.stdout.flush()

# 定义网络架构的辅助类
class MyNetwork:
    # 首先是常用的层
    def conv(self, data=None, name=None, n_filter=None, kernel=None):
        return (mx.sym.Convolution(data=data, name=name, kernel=(kernel, kernel),
            pad=((kernel-1)//2, (kernel-1)//2), num_filter = n_filter, cudnn_tune =
            config.auto_tune))

    def n(self, name=None):
        return self.name + name + str(self.layer)

    def act(self, data, name):

```

```

return mx.sym.Activation(data=data, name=self.n(name), act_type="relu")

def bn(self, data, name):
    return mx.sym.BatchNorm(data=data, name=self.n(name), fix_gamma=False)

# 常用的单元
def block(self, data=None, block_type=None, n_filter=None, kernel=None):
    if block_type == 'C':
        conv = self.conv(data=data, name=self.n("conv"), n_filter=n_filter,
                           kernel=kernel)
        return conv
    elif block_type == 'BA':
        bn = self.bn(data, "bn")
        act = self.act(bn, "act")
        return act
    elif block_type == 'CA':
        conv = self.conv(data=data, name=self.n("conv"), n_filter=n_filter,
                           kernel=kernel)
        act = self.act(conv, "act")
        return act
    elif block_type == 'BAC':
        bn = self.bn(data, "bn")
        act = self.act(bn, "act")
        conv = self.conv(data=act, name=self.n("conv"), n_filter=n_filter,
                           kernel=kernel)
        return conv
    elif block_type == 'R-BACBAC': # pre-act残差架构
        identity = data
        bn = self.bn(data, "bnA")
        act = self.act(bn, "actA")
        conv = self.conv(data=act, name=self.n("convA"), n_filter=n_filter,
                           kernel=kernel)
        bn2 = self.bn(conv, "bnB")
        act2 = self.act(bn2, "actB")
        conv2 = self.conv(data=act2, name=self.n("convB"), n_filter=n_filter,
                           kernel=kernel)
        return conv2 + identity

```

通过运用前面代码定义的单元，可用 for 语句定义一串单元。

```

# 可用于定义一串单元
def MyBlocks(self, data=None, blocks=None, block_type=None, name=None, n_filter=None, kernel=None):

    out = data

    for i in range(1, blocks+1):
        self.name = str(name)
        self.layer = i
        out = self.block(out, block_type, n_filter, kernel)
    return out

```

7.2.4 棋盘随机变换：symmetry.py

为节省内存，这里通过交换训练数据中的元素，实现 8 种棋谱的对称变换。

如果在 Python 中用循环实现这一点，速度会太慢。因此我们通过另一个程序直接预计算出所有赋值语句，得到很长但运行速度明显更快的代码：

```
#-*-coding:utf-8-*-
import random
import time
import numpy as np
import config

def apply_random_symmetry(b):

    # 对于每个局面...
    for i in range(0, len(b) / (config.head + 361)):

        k = i * (config.head + 361) # 找到其在数组中的偏移
        action = np.random.randint(1, 9) # 生成1-8的随机数，1代表不变

        if action==2: # 水平镜像
            b[k+0] = 18 - b[k+0] # 设置正确标签
            tmp = b[k+2]; b[k+2] = b[k+20]; b[k+20] = tmp
            【……后续内容很长，省略……】
            tmp = b[k+352]; b[k+352] = b[k+354]; b[k+354] = tmp
        elif action==3: # 垂直镜像
            b[k+1] = 18 - b[k+1] # 设置正确标签
            tmp = b[k+2]; b[k+2] = b[k+344]; b[k+344] = tmp
            【……后续内容很长，省略，包括action从4到8的处理……】
```

7.2.5 训练实例

训练代码的输出实例如下：

```
[pl 364] [pl validate data] [22:04:17] d:\program files (x86)\jenkins\workspace\
mxnet\mxnet\src\operator\./cudnn_algoreg-inl.h:106: Running performance tests
to find the best convolution algorithm, this can take a while... (setting env
variable MXNET_CUDNN_AUTOTUNE_DEFAULT to 0 to disable)
[pl 134] [pl 429] 0.00 : 1/2/5/10 10.68-17.79-28.74-38.35% ce 4.548 : lr 0.1000 : n/a
[pl 553] 0.00 : 1/2/5/10 20.85-32.04-47.94-58.46% ce 3.685 : lr 0.1000 : 6.28s
[pl 450] 0.01 : 1/2/5/10 24.84-35.66-52.06-63.67% ce 3.386 : lr 0.1000 : 6.31s
.....
=== [saved] : train 35.07% ce 2.692 : validate 37.12% ce 2.544 ===
.....
```

训练过程的性能曲线如图 7-3 所示，其中蓝线是训练准确率，红线是测试准确率，黑线是学习速率。

图中的训练过程如下：

- ❑ 至横坐标 3960：在 Tygem 数据集上训练，随机打乱棋谱，但不使用棋盘对称性，发现出现了明显的过拟合，即蓝线显著高于红线。
- ❑ 至横坐标 7140：使用棋盘的 8 种对称性，继续在 Tygem 数据集上训练，过拟合得以解决，蓝线降低到与红线相近的位置，然后两者继续上升。
- ❑ 至横坐标 9540：用较小的学习速率训练了一段时间，性能有一定提升。

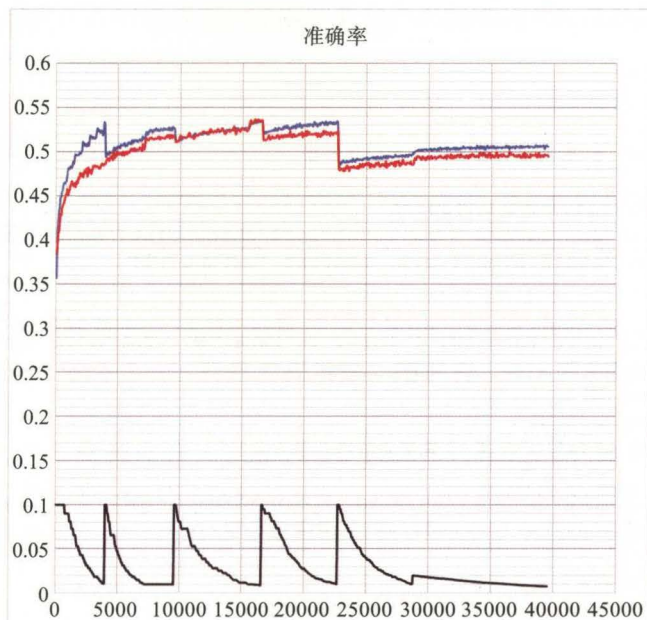


图 7-3 策略网络的训练过程

- 至横坐标 22680：分别用 KGS 数据集和 Tygem 数据集，再从较大的初始学习速率训练，性能继续提升。
- 至横坐标 28740：用 GoGoD 数据集，再从较大的初始学习速率训练，由于 GoGoD 中是职业棋手棋谱，难度更高，预测准确率明显降低。
- 至横坐标 39540：在 GoGoD 数据集上用更小的学习速率训练，以训练出更准确的网络。可见，神经网络的训练可以很灵活，即使中间出现了问题（过拟合），在解决问题后（加入数据增强），它就能自我纠正，无须重新从零训练。

经过与其他围棋程序的对战测试，最终得到的“39540 网络”的棋力确实是其中最高的。读者可在 <https://github.com/BlinkDL> 下载这个网络。

7.3 对弈实战

为进行人机对弈，建议实现 GTP（Go Text Protocol）协议，然后可通过 GoGui 等界面装载。由于这与深度学习的关系不大，在此我们简单介绍。

关于 GTP 协议，可参照 <http://www.lysator.liu.se/~gunnar/gtp/> 的内容，其中有示例程序。在实现 GTP 协议后，运行网络的方法，可参考下列代码：

```
#-*-coding:utf-8-*-
import mxnet as mx
import numpy as np
```

```

np.core.arrayprint._line_width = 120 # numpy输出行宽

N = 19 # 棋盘大小
features = 8 # 特征层数

first_run = True # 初次运行需要绑定和设置模组

# 载入模型，这里的39540是需装载的epoch
sym, arg_params, aux_params = mx.model.load_checkpoint("model//model", 39540)
module = mx.mod.Module(symbol=sym, context=[mx.gpu()])

# 输入的数据
data = [[[ 0 for x in range(N) ] for y in range(N)] for c in range(features)]
# 这里我们实验输入空棋盘
for x in range(0,N):
    for y in range(0,N):
        data[5][y][x] = 1 # 设置空点的特征层为全1

data=np.array([data]) # 变为1*8*19*19，即批大小为1
iter = mx.io.NDArrayIter(data=data) # 设置迭代器

if first_run: # 绑定和设置模组
    module.bind(data_shapes=iter.provide_data)
    module.set_params(arg_params, aux_params)
    first_run = False

pred = module.predict(iter).asnumpy() # 执行网络

# 输出结果
np.set_printoptions(formatter={'all':lambda x: str("%s" % str(int(round(x*100.0,0))).
    rjust(3))})
print(pred.reshape(N,N))

```

输出的例子如下，可见我们的网络在开局会优先选择左下角的星位：

```

[[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  4  0  0  0  0  0  0  0  0  0  0  0  0  4  0  0]
 [ 0  0  5  15  0  0  0  0  0  0  0  0  0  0  0  0  15  4  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  4  17  0  0  0  0  0  0  0  0  0  0  0  0  16  5  0]
 [ 0  0  0  5  0  0  0  0  0  0  0  0  0  0  0  0  4  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]

```

根据围棋的礼节，第1手应下在右上角，读者可加入代码处理。

如果希望用浏览器作为对弈界面,此时装载和使用网络的方法可参照 <https://withablink.coding.me/goPolicyNet/> 的源代码。

为快速测试棋力,可将我们的网络与其他程序对战,例如与 GnuGo 对战,如果读者的运行代码正确,会是百战百胜:

```
"C:\Program Files (x86)\GoGui\gogui.exe" -size 19 -computer-both -auto -program  
"C:\Program Files (x86)\GoGui\gogui-twogtp.exe -black "C:\gnugo\gnugo.  
exe --level 10 --mode gtp"" -white ""python play.py"" -games 5 -size 19  
-alternate -sgffile gnugo -verbose"
```

如果希望挑战更强的对手,可与 Leela 测试 (<https://www.sjeng.org/leela.html>), 注意下载 Leela 的 GTP 版本。由于 Leela 的棋力相当强,建议将 Leela 的 playout 数减少,否则会百战百败。

有兴趣的读者还可将程序连入 <http://www.yss-aya.com/cgos/19x19/standings.html> 与其他程序对战,连入方法见 <http://cgos.boardspace.net/>, 19 路围棋的服务器为 <http://yss-aya.com> 的 6819 端口。建议先写好终盘清理死子的代码,否则在 Tromp-Taylor 规则下会吃亏。

生成式对抗网络：GAN

8.1 GAN 的起源故事

在几年前，深度学习的热门话题是图像分类，即，输入一张图 x ，深度网络可输出其中物体的类别 c ，如图 8-1 所示。

而近年来，深度学习的热门话题是生成模型，其中的代表是 GAN。它可实现相反的事情，例如，对于指定的类别 c ，GAN 可无穷无尽地自动生成真实而多变的此类别的图像 x 。

这听上去更神奇，但 GAN 的实际方法很简单，甚至简单得有些出人意料。



图 8-1 分类与 GAN 的对比

著名物理学家费曼曾说过：“What I cannot create, I do not understand.”（如果我不能创造某事物，就说明我还没有理解这样事物。）换言之，创造是理解的重要前提。

但如何让深度网络学会创造出高质量的图像？这是 2014 年，加拿大 Montréal 大学的一名博士生 Goodfellow 思考的问题。

Goodfellow 师从“深度学习三巨头”之一的 Bengio, 对于此前的研究非常了解: 能生成图像的神经网络, 经典的例子是 1985 年的玻尔兹曼机 (Boltzmann machine) 和 1987 年的自编码器 (Auto-Encoder, AE)。其中前者偏向理论研究, 难以在实际中使用。后者生成的图像较为模糊, 人一眼就可看出其虚假性。

下面的故事听上去有些戏剧性, 但的确属实。某天, Goodfellow 和同学在酒吧里为另一位同学送行。他和同学谈起了生成模型, 突然产生了一个想法:

- 令负责生成图像的深度网络为 G (Generator, 生成器)。
 - G 的输入称为 z , 可以是指定的编码, 也可以是随机噪声, 例如 100 个高斯分布的随机数。
 - G 的输出 $G(z)$ 是图像, 例如 64×64 的彩色图像。
- 用另一个深度网络 D (Discriminator, 判别器), 负责判别 G 生成的图像的真实性。
 - D 的输入称为 x , 是待判定的图像。
 - D 的输出 $D(x)$ 是对图像的真实度的判定, $D(x)=1$ 代表图像完全真实, $D(x)=0$ 代表图像完全虚假。
- 由于深度网络很擅长分辨图像, 如果 G 生成的图像足以骗过 D , 可能就说明 G 生成的图像质量很高, 也许足以同样骗过人类, 如图 8-2 所示。
- D 的目标是尽量辨认出 G 生成的图像 $G(z)$ 与真实图像的区别。即, $D(G(z))$ 应趋于 0。同时, 如果 x 是真实图像, 那么 $D(x)$ 应趋于 1。
- G 的目标是尽量生成 D 无法分辨真假的图像。即, 尽量让 $D(G(z))$ 趋于 1。
- G 和 D 可随着训练不断进步。

于是 Goodfellow 在酒吧告诉同学, 这个办法肯定行得通。不过, 他的同学完全不相信。确实, 这听上去似乎过于美好, G 和 D 完全有可能很快就陷入局部最优, 停滞不前。

然而, 深度学习很神奇, 许多听上去不可思议的方法竟然可以工作。回到家中后, Goodfellow 立刻开始编写代码。在午夜时分, 他惊喜地发现, 第一版程序, 竟然就成功了。

这个由 G 和 D 构成的网络架构, 称为生成式对抗网络 (Generative Adversarial Networks, GAN)。必须说, Goodfellow 很幸运, 因为后续的研究人员发现, 最初的 GAN 往往需要调整不少超参数, 才可能实现较为稳定的训练。

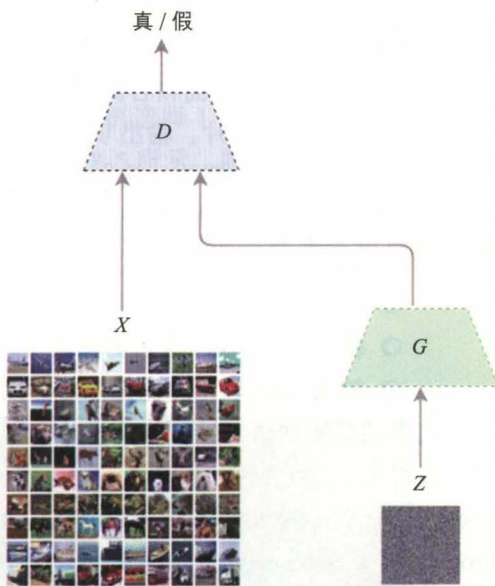


图 8-2 GAN 的基本架构

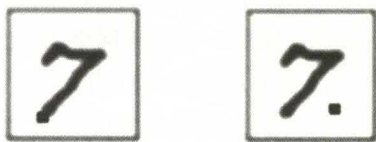
8.2 GAN 的基本原理

8.2.1 生成模型：从图像到编码，从编码到图像

生成模型属于无监督学习，只需大量真实图像作为训练数据，就能生成更多在人看来真实的图像。

这里的难点是，在像素的级别很难实现这个目标。举例：

- 如果 X 和 Y 是两张真实图像，那么通过像素点对点运算得到的 $(X+Y)/2$ 往往并不是真实图像。
- 对于真实图像 X ，令 A 和 B 为两张与 X 的像素距离（例如 MSE）相同的图像，那么 A 和 B 有可能在人类看来真实性完全不同。例如，图 8-3 左右两张图都是将数字 7 的图像改变 1 个像素，但前图看上去仍然真实，而后图就看上去明显是多了 1 个点。



为解决这个问题，可在语义层面实现这个目标，将图像 x 变为具有语义意义的编码 z ，再从编码 z 得到图像 x 。举例说明：

- 给定一个人的正面和右侧面照片，我们希望找到一个编码（encoder）网络：
 - 输入正面照片，输出 0。
 - 输入右侧面照片，输出 1。
- 同时，我们希望找到一个解码（decoder）网络：
 - 输入 0，输出正面照片。
 - 输入 1，输出右侧面照片。
 - 输入 0.5，输出半右侧面照片。
 - 输入 -1，输出左侧面照片。以此类推。

这里的解码网络，就是一个生成模型。而且实际上我们也能做到这一点，例如《Beyond Face Rotation: Global and Local Perception GAN for Photorealistic and Identity Preserving Frontal View Synthesis》^①显示，通过使用 GAN 的思想，只输入 1 张侧面照片，就可生成颇为真实的正面照片，如图 8-4 所示。

中间是输入的侧面照片，左边是深度网络生成的正面照片，右边是实际的正面照片。可以看到，网络的猜测相当接近真实，人类也不一定猜测得这样准确。

对于更一般的情况，我们希望把照片编码为更复杂的

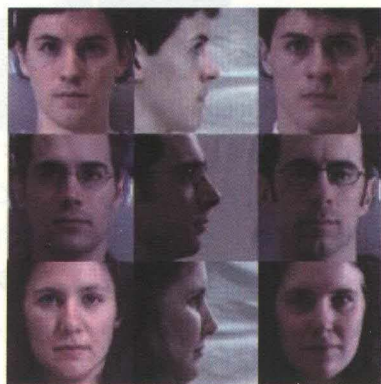


图 8-4 由侧面照片生成正面照片

^① 地址为 <https://arxiv.org/abs/1704.04086>。

一串数字,例如用1个数字代表面部的左右角度,1个数字代表上下角度,1个数字代表面部的大小,1个数字代表男女,1个数字代表年龄,1个数字代表笑的程度,等等。

这里的问题是:

1) 如果数据集有标签,例如已有每张图像的编码,那么训练编码网络很简单,就像训练图像分类网络。但如果数据集无标签,电脑是否可自动发现左右角度、上下角度等是图像的重要语义因素?

2) 在训练解码网络时,如果只要求它满足“输入0,输出正面照片”“输入1,输出右侧面照片”,那么,它有可能变成“输入 x ,输出正面照片 $*(1-x)$ +右侧面照片 $*x$ ”,即只是在像素层面解决问题,而不是在语义层面解决问题。

这些问题听上去很难解决,然而目前采用深度卷积网络的生成模型,有能力自动解决这些问题,因为深度卷积网络具有相当强的从像素提取语义的能力,正如它此前在图像分类问题中所显示的。

8.2.2 GAN 的基本效果

在此我们展示的图像主要来自于2015年的DCGAN[⊖],并不是最新最好的效果,但以看出GAN的能力。后文我们会介绍DCGAN的详细训练过程和代码。

首先,GAN可自动生成无穷无尽的图像,因为每随机选择1个编码作为G的输入,就可以输出1张新图像。GAN生成的明星照片效果如图8-5所示。效果不错,不过有一些瑕疵。是否可改进这些瑕疵?

在此的最新进展由nVidia在2017年10月发布,生成的明星照片非常清晰和真实,如图8-6所示。



图 8-5 由 DCGAN 生成的明星照片



图 8-6 由 nVidia 方法生成的高质量明星照片

我们会在后文介绍它的原理。

⊖ 地址为 <https://arxiv.org/abs/1511.06434>。

需要说明，图 8-5 和图 8-6 中采用的 CelebA 数据集较为简单，容易夸大模型的效果。如果在更复杂的数据集上，目前 GAN 的效果会明显更差，例如，如果将 nVidia 的方法应用于 LSUN 数据集，生成猫图像，效果仍然颇为糟糕，如图 8-7 所示。

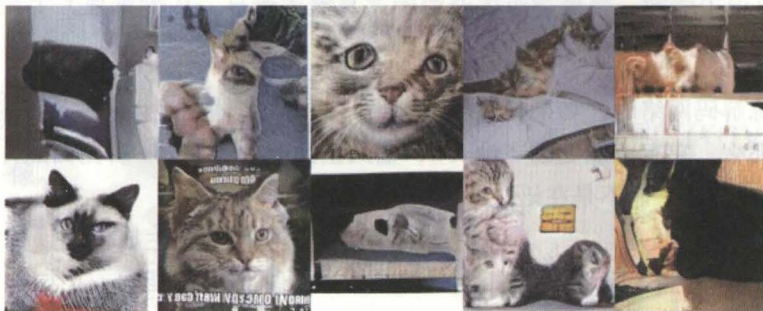


图 8-7 将 nVidia 方法用于更复杂数据集的效果

GAN 也可生成 3D 模型，因为我们可将 3D 模型看作由 3 维空间中的体素（voxel）组成，如图 8-8[⊖]所示。

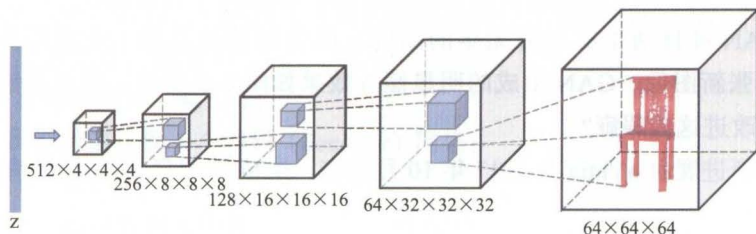


图 8-8 用 GAN 生成立体模型

回到 DCGAN，再看编码 z 的插值，目标是将两张图像的语义进行插值，具体方法如下：

- 在训练完 GAN 后，对于指定的两张图像，找到它们在 GAN 中的编码。由于普通 GAN 只有 G 负责解码，并没有编码的部分，因此这需要少量额外计算。
- 将两张图像的编码线性插值（有时用球形插值会得到更佳效果）。
- 用 GAN 生成这些中间编码对应的图像。

图 8-9 中每行的左右是我们指定的两张图像，中间是 GAN 的插值结果。可见，房间中的摆设都会自动实现较为平滑的变换。中间的图像虽然有一些瑕疵，但考虑到整个训练过程是全自动的，图像都没有标签，语义都是网络自动发现的，所以这个效果已是出人意料地好了。

⊖ 图片来源：<http://3dgan.csail.mit.edu/>。



图 8-9 用 GAN 对图像的语义进行插值

再看编码的改变。例如，我们可找到“在笑的女人”“不笑的女人”“不笑的男人”等的平均编码，然后对编码进行简单的加减法计算，要求 GAN 输出新编码所对应的图像，如图 8-10 所示。

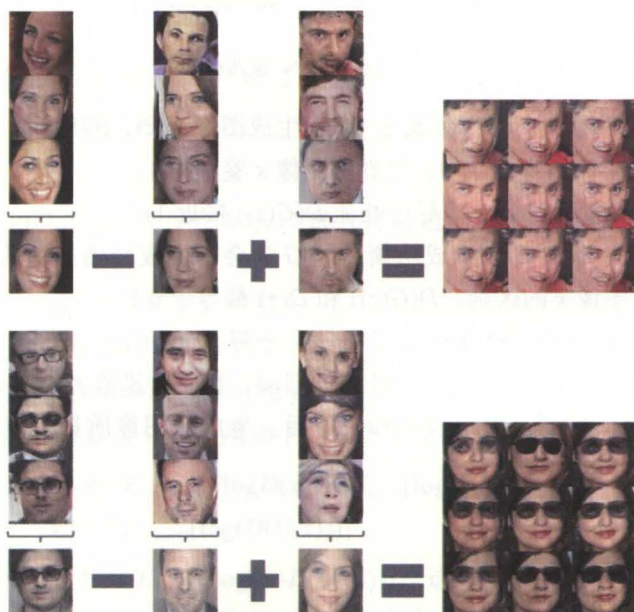


图 8-10 用 GAN 改变图像的语义

可见，GAN 可成功实现这样的语义等式：

“在笑的女人” - “不笑的女人” + “不笑的男人” = “在笑的男人”
 “戴眼镜的男人” - “不戴眼镜的男人” + “不戴眼镜的女人” = “戴眼镜的女人”

这进一步说明 GAN 可自动找到具有语义意义的编码，再将编码变为相应的图像。

8.2.3 GAN 的训练方法

GAN 的训练基本思想如图 8-11 所示。

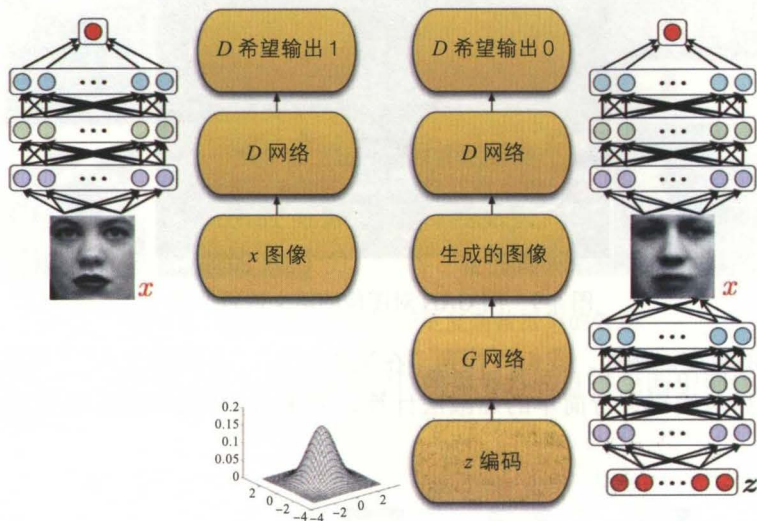


图 8-11 GAN 的训练

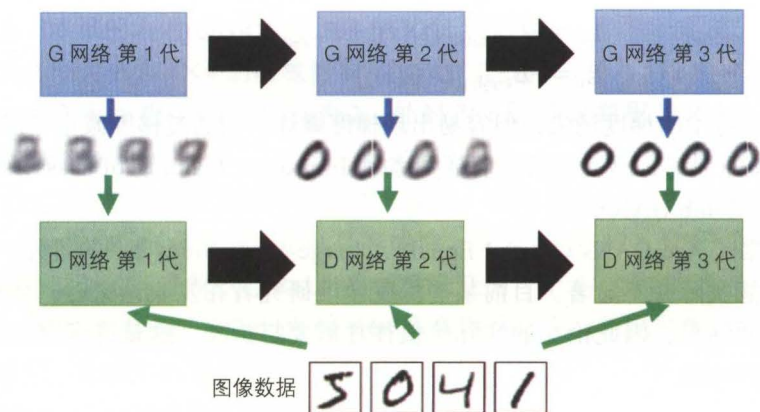
令 x 为真实图像数据集。对于编码 z ， G 将生成图像 $G(z)$ ，然后：

- D 的目标是将 $G(z)$ 鉴别为 0，将真实图像 x 鉴别为 1。
- G 的目标是尽量骗过 D ，即尽可能让 $D(G(z))$ 接近 1。
- 理论上，最终 D 和 G 会达成均衡，即 G 学会了生成栩栩如生的图像， D 无法分辨 $G(z)$ 和真实图像 x 的区别， $D(G(z))$ 和 $D(x)$ 都等于 0.5。
 - 但在实际训练中， D 往往会很快学会分辨 $G(z)$ 和真实图像 x 的区别，虽然有时会被 G 骗过，但很快又会找到 G 的漏洞，恢复辨别能力。
 - 另一方面， G 在一次次成功和失败后，生成的图像质量会越来越高，如图 8-12 所示。

具体训练流程是：

- 1) 随机选取真实图像 x 。
- 2) 将 x 输入 D ，得到 $D(x)$ 。
- 3) 希望 $D(x)=1$ ，获得反向梯度，保存备用。
- 4) 由随机采样生成 z ，例如令 z 为 100 维的 $\{z_1, z_2, \dots, z_{100}\}$ ，其中 z_i 是标准差为 1 的正态分布的随机数。
- 5) 将 z 输入 G ，生成 $G(z)$ 。

- 6) 将 $G(z)$ 输入 D , 得到 $D(G(z))$ 。
- 7) 希望 $D(G(z))=0$, 获得反向梯度, 与之前 D 的梯度相加, 训练 D 。
- 8) 将 $G(z)$ 再次输入 D , 得到新的 $D(G(z))$ 。
- 9) 希望 $D(G(z))=1$, 获得对于输入的梯度, 反向传入 G , 训练 G 。
- 10) 重复此过程。

图 8-12 GAN 中 G 和 D 的进化

下面是数学细节, 不熟悉的读者可跳过。在 GAN 的训练过程中, 优化目标可写成:

$$\min_G \max_D V(D, G) = E_{x \sim p_{\text{data}}(x)} [\log D(x)] + E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

在此简单解释:

- $\min_G \max_D V(D, G)$ 代表首先要求 D 最大化右边的式子。然后要求 G 最小化右边的式子。
- 右边式子中的 \log 来自于交叉熵损失。
- $x \sim p_{\text{data}}(x)$ 代表 x 符合真实图像的统计分布 p_{data} , 即 x 属于真实图像。
- $x \sim p_z(z)$ 代表 z 符合编码的统计分布 p_z , 即 z 为从编码的统计分布采样的随机数。

在实际程序中, D 和 G 会分别使用如下的损失 (优化的目标是 minimized 这些损失):

$$L_D = -E_{x \sim p_{\text{data}}(x)} [\log D(x)] - E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

$$L_G = -E_{z \sim p_z(z)} [\log(D(G(z)))]$$

注意其中 G 的损失不是 $E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$ 。这是为了改善梯度, 因为目标是让 $D(G(z))$ 接近 1, 而不是让 $D(G(z))$ 远离 0。让我们检验:

- D 希望看到的是 $D(x)=1$, $D(G(z))=0$, 此时 $L_D=0$ 。
- D 希望避免的是 $D(x)=0$, $D(G(z))=1$, 此时 $L_D=\infty$ 。
- G 希望看到的是 $D(G(z))=1$, 此时 $L_G=0$ 。
- G 希望避免的是 $D(G(z))=0$, 此时 $L_G=\infty$ 。

可见， D 和 G 最希望看到的都是最小化各自的损失，符合损失的定义。

最后，根据导数的性质，如果我们希望让网络 X 最小化 $A+B$ ，且 A 和 B 独立，那么应先最小化 A ，得到梯度，再最小化 B ，得到梯度，再将两个梯度相加，用于训练网络 X 。这就是为何在训练 D 时需要存储梯度，并与后续的梯度相加。

在 GAN 的后续研究中，研究人员提出了多种不同损失，例如 Wasserstein 损失 (<https://arxiv.org/abs/1701.07875>)：

$$L_D = -E_{x \sim p_{\text{data}}(x)}[D(x)] + E_{z \sim p_z(z)}[D(G(z))]$$

$$L_G = -E_{z \sim p_z(z)}[D(G(z))]$$

它可改善训练中的梯度消失，但容易出现梯度爆炸，需通过梯度截止等技巧解决。

对于 GAN 的训练，感兴趣的读者还可参阅 Improved Training of Wasserstein GANs[⊖]，BEGAN[⊗]，Coulomb GAN[⊕]等。

不过，根据《Are GANs Created Equal? A Large-Scale Study》[Ⓜ]的研究，各种 GAN 间的性能差异可能实际并不显著。目前某些深度学习研究者在发表论文时，会有意无意地选出对自己有利的结果，因此论文的实际价值往往需要打折扣。这是许多资深研究者认为学术界需要改善的问题。

8.3 实例：DCGAN 及训练过程

8.3.1 网络架构

经典且易于训练的 GAN 架构是在《Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks》[Ⓜ]中提出的 DCGAN (Deep Convolutional GAN)，它的 G 和 D 网络都采用全卷积架构，其中 G 网络的结构如图 8-13 所示。

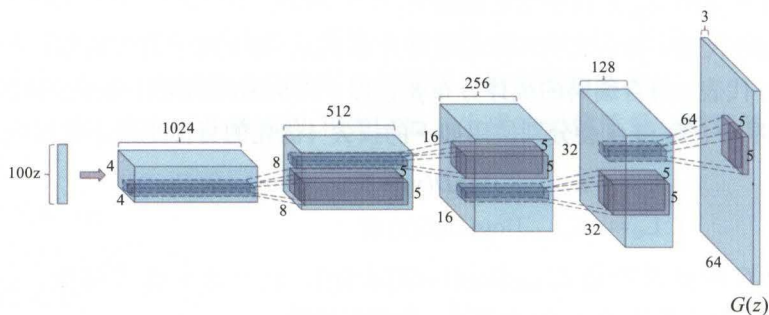


图 8-13 DCGAN 中 G 的架构

⊖ 地址为 <https://arxiv.org/abs/1704.00028>。

⊗ 地址为 <https://arxiv.org/abs/1703.10717>。

⊕ 地址为 <https://arxiv.org/abs/1708.08819>。

Ⓜ 地址为 <https://arxiv.org/abs/1711.10337>。

Ⓜ 地址为 <https://arxiv.org/abs/1511.06434v2>。

在 G 网络中, 需将图像不断放大, 这可通过转置卷积。注意 $X \times X$ 的图像经过大小为 4×4 , 步长为 2, 外衬为 1 的转置卷积可变为 $2X \times 2X$ 的图像, 即边长扩为原来的 2 倍, 这是在 G 网络中常常会用到的一种转置卷积。

在 D 网络中, 需将图像不断缩小, 可通过对称的方法, 即大小为 4×4 , 步长为 2, 外衬为 1 的卷积, 将 $2X \times 2X$ 的图像变为 $X \times X$ 的图像。

DCGAN 中 G 的架构是 (为简明起见, 下面的描述省略了 BN 层):

- 输入为 100 维的编码 z , 可看作 100 个 1×1 的通道。
- 经过 512 个大小为 4×4 的转置卷积与 ReLU 非线性, 变为 512 个 4×4 的通道。
- 经过 256 个大小为 4×4 , 步长为 2, 外衬为 1 的转置卷积与 ReLU 非线性, 变为 256 个 8×8 的通道。
- 经过 128 个类似的转置卷积与 ReLU 非线性, 变为 128 个 16×16 的通道。
- 经过 64 个类似的转置卷积与 ReLU 非线性, 变为 64 个 32×32 的通道。
- 经过 3 个类似的转置卷积与 ReLU 非线性, 变为 3 个 64×64 的通道。
- 经过 tanh 非线性, 即为输出的 64×64 彩色图像。在 DCGAN 中, 将彩色图像的像素值都归一化到 $[-1, 1]$ 区间。

而 D 的架构与 G 的架构基本是对称的:

- 输入为 64×64 彩色图像, 即 3 个 64×64 的通道。
- 经过 64 个大小为 4×4 , 步长为 2, 外衬为 1 的卷积与 Leaky ReLU 非线性, 变为 64 个 32×32 的通道。DCGAN 使用 Leaky ReLU (即当 $x < 0$ 时输出 $0.2 \times x$) 以改善梯度消失。
- 经过 128 个类似的卷积与 Leaky ReLU 非线性, 变为 128 个 16×16 的通道。
- 经过 256 个类似的卷积与 Leaky ReLU 非线性, 变为 256 个 8×8 的通道。
- 经过 512 个类似的卷积与 Leaky ReLU 非线性, 变为 512 个 4×4 的通道。
- 经过 1 个 4×4 的卷积, 变为 1 个 1×1 的通道。
- 摊平并经过 sigmoid 非线性, 即对于输入图像的真假概率的判定值。

在具体训练时, 研究人员发现 Adam 优化器的效果较好, 且需使用较小的学习速率 (如 0.0002), 并将 Adam 优化器的 beta1 动量从 0.9 减少为 0.5, 以提高训练的稳定性。

8.3.2 训练代码

请读者在 Jupyter Notebook 运行此代码, 以看到运行过程中生成的图像。

这里的输入图像为 64×64 彩色图像, 例如可采用 CelebA 数据集^①。

首先是常规的定义和网络的参数。

```
from __future__ import print_function
import mxnet as mx
```

① 地址为 <http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>。

```

import numpy as np
import time
from matplotlib import pyplot as plt
from datetime import datetime

ctx = mx.gpu(0)

stamp = datetime.now().strftime('%Y_%m_%d-%H_%M') # 设置输出时间格式
plt.rcParams['figure.figsize'] = (6.6, 6.6) # 增大图像的显示尺寸

img_path = 'data/celeba.rec' # 训练图像
img_size = 64 # 图像尺寸
nc = 3 # 图像的通道数, 3代表彩色

Z = 100 # 编码维数
batch_size = 64 # 批大小
ndf = 64 # D网络的大小参数
ngf = 64 # G网络的大小参数

# 如需继续训练可设置这里
loadG = None # 例如"G_2017_09_11-07_38"
loadD = None # 例如"D_2017_09_11-07_38"

check_point_period = 1 # 每隔多少epoch保存模型

# 训练参数
lr_G = 0.0002
lr_D = 0.0002
beta1_G = 0.5
beta1_D = 0.5
wd_G = 0
wd_D = 0

leaky_slope = 0.2 # Leaky ReLU的负轴泄露程度

定义网络所需的辅助函数。

def Leaky(s, name):
    return mx.sym.LeakyReLU(s, act_type='prelu', slope=leaky_slope, name=name)

def BatchNorm(s, name): # 设置fix_gamma为True有可能改善模式坍塌
    return mx.sym.BatchNorm(s, name=name, fix_gamma=True, eps=1e-5 + 1e-12)

def Upsize2x(s, name, num_filter):
    return mx.sym.Deconvolution(s, name=name, kernel=(4,4), stride=(2,2),
                                pad=(1,1), num_filter=num_filter)

def Downsize2x(s, name, num_filter):
    return mx.sym.Convolution(s, name=name, kernel=(4,4), stride=(2,2),
                              pad=(1,1), num_filter=num_filter)

def Upsize2x_BN_Act(s, name, num_filter):
    s = Upsize2x(s, name=name, num_filter=num_filter)
    s = BatchNorm(s, name=name+'_bn')
    s = Leaky(s, name=name+'_act')
    return s

```

```
def Downsize2x_BN_Act(s, name, num_filter):
    s = Downsize2x(s, name=name, num_filter=num_filter)
    s = BatchNorm(s, name=name+'_bn')
    s = Leaky(s, name=name+'_act')
    return s
```

定义 G 和 D 网络。

```
def make_dcgan_sym(ngf, ndf, nc):

    Gnet = mx.sym.Variable('rand')

    Gnet = mx.sym.Deconvolution(Gnet, name='g1', kernel=(4,4), num_filter=ngf*8)
    # 增大至4x4
    Gnet = BatchNorm(Gnet, name='g1_bn')
    Gnet = Leaky(Gnet, name='g1_act')

    Gnet = Upsize2x_BN_Act(Gnet, name='g2', num_filter=ngf*4) # 增大至8x8
    Gnet = Upsize2x_BN_Act(Gnet, name='g3', num_filter=ngf*2) # 增大至16x16
    Gnet = Upsize2x_BN_Act(Gnet, name='g4', num_filter=ngf) # 增大至32x32

    Gnet = Upsize2x(Gnet, name='g5', num_filter=nc) # 增大至64x64
    Gnet = mx.sym.Activation(Gnet, act_type='tanh', name='gout') # 经过tanh后输出,
    # 范围为-1到1

    #####

    Dnet = mx.sym.Variable('data')

    Dnet = Downsize2x(Dnet, name='d1', num_filter=ndf) # 缩小至32x32
    Dnet = Leaky(Dnet, name='d1_act')

    Dnet = Downsize2x_BN_Act(Dnet, name='d2', num_filter=ndf*2) # 缩小至16x16
    Dnet = Downsize2x_BN_Act(Dnet, name='d3', num_filter=ndf*4) # 缩小至8x8
    Dnet = Downsize2x_BN_Act(Dnet, name='d4', num_filter=ndf*8) # 缩小至4x4

    Dnet = mx.sym.Convolution(Dnet, name='d5', kernel=(4,4), num_filter=1) # 缩小至1x1
    Dnet = mx.sym.Flatten(Dnet)
    Dnet = mx.sym.LogisticRegressionOutput(Dnet, label=mx.sym.Variable('label'),
    name='dloss') # 做Logistic回归

    # 研究人员认为G的输出层和D的输入层可能不应该加BN, 读者也可自行实验

    return Gnet, Dnet
```

定义提供 z 和 x 的迭代器。

```
# 提供编码z的迭代器
class RandIter(mx.io.DataIter):
    def __init__(self, batch_size, ndim):
        self.batch_size = batch_size
        self.ndim = ndim
        self.provide_data = [('rand', (batch_size, ndim, 1, 1))]
        self.provide_label = []

    def iter_next(self):
        return True
```



```

def getdata(self): # 从随机采样得出z
    return [mx.nd.random_normal(0.0, 1.0, shape=(self.batch_size, self.ndim, 1, 1))]

# 提供图像x的迭代器
class ImageIter(mx.io.DataIter):
    def __init__(self, path, batch_size, data_shape):
        self.internal = mx.io.ImageRecordIter(
            path_imgrec = path,
            data_shape = data_shape,
            batch_size = batch_size,
            shuffle = True,
            resize = img_size,
            min_crop_size = img_size,
            max_crop_size = img_size,
            min_img_size = img_size,
            max_img_size = img_size,
            rand_crop = False,
            rand_mirror = True)
        self.provide_data = [('data', (batch_size,) + data_shape)]
        self.provide_label = []

    def reset(self):
        self.internal.reset()

    def iter_next(self):
        return self.internal.iter_next()

    def getdata(self):
        data = self.internal.getdata()
        data = data * (2.0/255.0) - 1.0 # 将图像数据归一化到-1与1之间
        return [data]

```

用于输出 GAN 生成的图像以供查看的函数。

```

# 用于输出图像的辅助函数
def fill_buf(buf, i, img, shape):
    n = buf.shape[0]/shape[1]
    m = buf.shape[1]/shape[0]
    sx = (i%m)*shape[0]
    sy = (i/n)*shape[1]
    buf[sy:sy+shape[1], sx:sx+shape[0], :] = img

# 用于输出图像的函数
def visual(title, X):
    X = X.transpose((0, 2, 3, 1))
    X = np.clip((X+1.0)*255.0/2.0, 0, 255).astype(np.uint8)
    n = np.ceil(np.sqrt(X.shape[0]))
    buff = np.zeros((int(n*X.shape[1]), int(n*X.shape[2]), int(X.shape[3])),
        dtype=np.uint8)
    for i, img in enumerate(X):
        fill_buf(buff, i, img, X.shape[1:3])
    plt.imshow(buff)
    plt.title(title)
    plt.show()

```

训练的主程序。首先初始化网络和模组等。

```

if __name__ == '__main__':

    print('Loading...')

    # 定义网络架构和迭代器
    symG, symD = make_dcgan_sym(ngf, ndf, nc)
    train_iter = ImageIter(img_path, batch_size, (nc, img_size, img_size))
    rand_iter = RandIter(batch_size, Z)

    label = mx.nd.zeros((batch_size, ), ctx=ctx) # 训练用的标签

    # ===== 定义G网络模组 =====

    modG = mx.mod.Module(symbol=symG, data_names=('rand',), label_names=None,
                          context=ctx)
    modG.bind(data_shapes=rand_iter.provide_data, inputs_need_grad=True)

    if loadG: # 从文件读入参数, 或初始化参数
        modG.init_params(initializer=mx.init.Load(loadG))
    else:
        modG.init_params(initializer=mx.init.MSRAPrelu(slope=leaky_slope))

    modG.init_optimizer(
        optimizer='adam',
        optimizer_params={
            'learning_rate': lr_G,
            'wd': wd_G,
            'betal': betal_G,
        })

    # ===== 定义D网络模组 =====

    modD = mx.mod.Module(symbol=symD, data_names=('data',), label_names=('label',),
                          context=ctx)
    modD.bind(data_shapes=train_iter.provide_data,
              label_shapes=[('label', (batch_size,))],
              inputs_need_grad=True)

    if loadD: # 从文件读入参数, 或初始化参数
        modD.init_params(initializer=mx.init.Load(loadD))
    else:
        modD.init_params(initializer=mx.init.MSRAPrelu(slope=leaky_slope))

    modD.init_optimizer(
        optimizer='adam',
        optimizer_params={
            'learning_rate': lr_D,
            'wd': wd_D,
            'betal': betal_D,
        })

    定义性能指标, 开始训练。

    # ===== 定义性能指标 =====
    def facc(label, pred):
        pred = pred.ravel()
        label = label.ravel()

```

```

return ((pred > 0.5) == label).mean()

mReal = mx.metric.CustomMetric(facc)
mFake = mx.metric.CustomMetric(facc)

# ===== 开始训练 =====

print('Training...')
start_time = time.time()

for epoch in range(500): # 训练epoch数

    train_iter.reset()
    for t, batch in enumerate(train_iter):

        # 将batch中的x送入D, 要求D输出1
        label[:] = 1 # 设置输出标签为1
        batch.label = [label]
        modD.forward(batch, is_train=True) # 将batch送入D
        modD.update_metric(mReal, [label]) # 更新性能指标
        modD.backward() # 反向传播
        # 将梯度保存待用
        gradD1 = [[grad.copyslot(grad.context) for grad in grads] for grads in
                  modD._exec_group.grad_arrays]

        # 将rbatch中的z送入G
        rbatch = rand_iter.next() # 需手工运行rand_iter, 获得一批z
        modG.forward(rbatch, is_train=True) # 送入G
        outG = modG.get_outputs() # 获得输出

        # 将G生成的x送入D, 要求D输出0
        label[:] = 0
        modD.forward(mx.io.DataBatch(outG, [label]), is_train=True) # 送入D
        modD.update_metric(mFake, [label]) # 更新性能指标
        modD.backward() # 反向传播

        # 将D在两种情况的梯度相加
        for i in range(len(gradD1)):
            for j in range(len(gradD1[i])):
                modD._exec_group.grad_arrays[i][j] += gradD1[i][j] # 加上之前的梯度
        modD.update() # 更新D的参数

        # 调整G的参数, 使D(G(z))更接近1
        label[:] = 1
        # is_train会影响BN, 因此这里将is_train关闭, 否则会重复更新BN的统计信息
        modD.forward(mx.io.DataBatch(outG, [label]), is_train=False) # 先对D做前向传播
        modD.backward() # 反向传播, 获得D的梯度
        modG.backward(modD.get_input_grads()) # 将D对于输入的梯度送入G反向传播
        modG.update() # 更新G的参数

```

在每完成 1 个 epoch 后, 输出训练信息和生成的图像。

```

# 输出epoch的信息
print('epoch:', epoch, 'real:', mReal.get()[1], 'fake:', mFake.get()[1],
      'time:', time.time()-start_time)
start_time = time.time()
mReal.reset()

```



```

mFake.reset()

visual('gout', outG[0].asnumpy()) # 输出epoch的图像

if check_point_period > 0 and epoch % check_point_period == (check_point_
    period - 1): # 保存网络参数
    modG.save_params('G_%s-%04d.params'%(stamp, epoch))
    modD.save_params('D_%s-%04d.params'%(stamp, epoch))

```

在此需要解释的是, 如何调整 G 的参数使 $D(G(z))$ 接近 1。回想梯度的定义, 是指 LOSS 对于变量的偏导。而 `modD.get_input_grads()` 是 $\text{LOSS}=D$ 对于输入的梯度, 即 LOSS 对于 D 的输入的偏导, 即 LOSS 对于 G 的输出的偏导, 将其带入 G 反向传播, 即可获得 LOSS 对于 G 的参数的偏导, 即 G 的梯度。

网络的正常运行输出, 和出现模式坍塌 (mode collapse) 时的输出, 如图 8-14 所示。

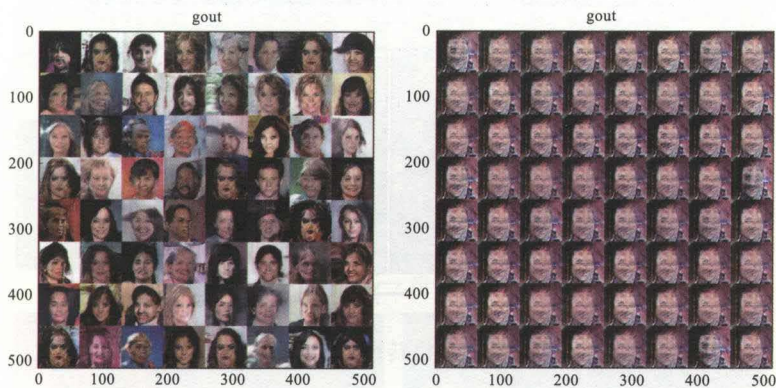


图 8-14 网络的正常输出和异常输出

模式坍塌是训练 GAN 网络的主要难点, 即 G 突然只会生成少量固定的图像, 生成的图像失去了多样性。这个现象可用 Inception Score[⊖] 衡量。

8.4 GAN 的更多架构和应用

作为近年来深度学习中最热门的话题之一, GAN 的架构和应用层出不穷, 例如在 <https://github.com/hindupuravinash/the-gan-zoo> 列举出了上百种不同的 GAN 设计。在此我们介绍其中的部分著名应用。

8.4.1 图像转移: CycleGAN 系列

首先是 CycleGAN[⊖]。给定两组图像, 它可以自动学会两组图像之间的风格或语义差异, 并有能力将新的图像进行风格或语义的转移。

[⊖] 地址为 <https://arxiv.org/abs/1606.03498>。

[⊖] 地址为 <https://arxiv.org/pdf/1703.10593.pdf>。

举例说明，如果其中一组图像中是风景照，另一组图像中是印象派画家莫奈的画作，无须任何额外的标签和对应关系（它们可以是在描绘完全不同的场景），CycleGAN 就可自动学会将风景照转化为莫奈风格的画作，以及将莫奈风格的画作转化为风景照，效果相当真实，如图 8-15 所示。

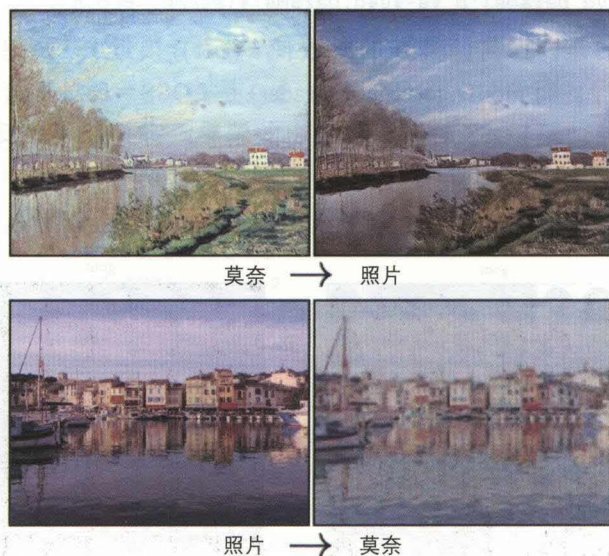


图 8-15 风格转换：莫奈与照片

同样，CycleGAN 可无师自通，学会将照片变换为不同画家风格的画作，如图 8-16 所示。

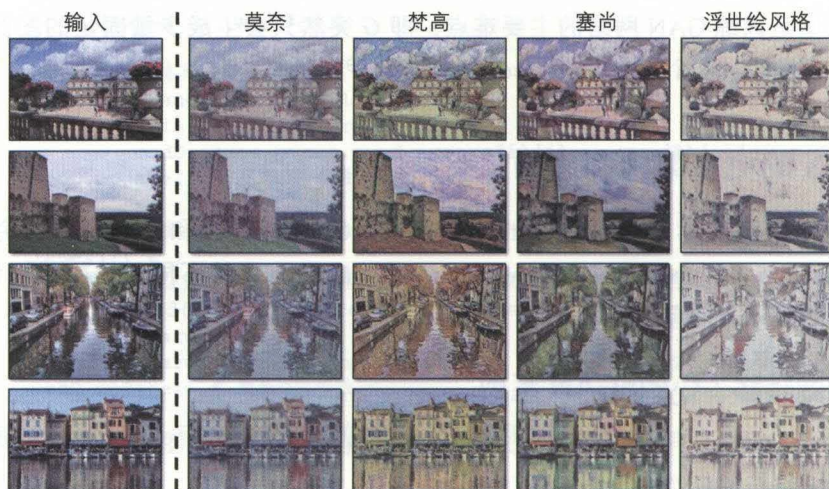


图 8-16 风格转换：模仿大师

这属于前文提过的风格转移, CycleGAN 的优势是效果往往更好。

更多有趣的例子, 包括自动将图像中的斑马变成马, 将马变成斑马。或将夏天的风景照变为冬天的风景照, 将冬天的风景照变为夏天的风景照。还可在苹果和橙子之间转换, 效果都很真实, 如图 8-17 所示。



图 8-17 CycleGAN 的更多效果

通过学习手机拍摄的照片和单反相机拍摄的照片, 学会做背景虚化, 如图 8-18 所示。



图 8-18 通过 CycleGAN 实现背景虚化

自动将草绘变为真实图像, 将真实图像变为草绘, 如图 8-19 所示。



图 8-19 通过 CycleGAN 将草绘变为真实图像

甚至可实现图像分割（效果不如专门的图像分割网络），还能从分割数据还原出原始图像（这是生成模型的独门绝技），如图 8-20 所示。

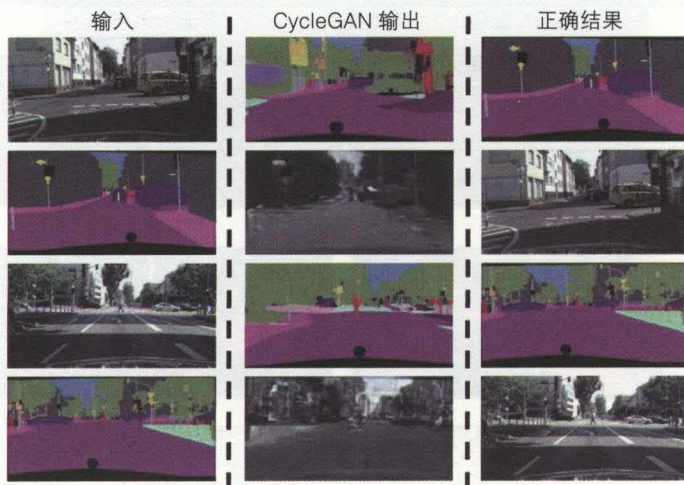


图 8-20 通过 CycleGAN 分割和生成街景

在 <https://junyanz.github.io/CycleGAN/> 还可看到更多具有创意的应用。例如，将左边的古代地图转变为右边的现代地图和卫星地图，如图 8-21 所示。

将知名美剧《权力的游戏》中的角色变为洋娃娃，效果相当好，如图 8-22 所示。

将猫变为狗（来自 https://qiita.com/itok_msi/items/b6b615bc28b1a720afd7 的改进架构），如图 8-23 所示。

CycleGAN 还可改变图像中人物的性别，自动美颜，自动“丑颜”（将美颜效果去掉），等等，可谓只有想不到，没有做不到。

例如，一位爱吃拉面的日本网友发现，它甚至可将拉面变为人，人变为拉面，如图 8-24 所示。如果仔细观察，会发现竟然有一定道理。

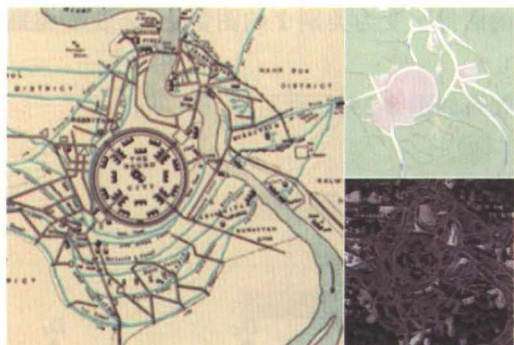


图 8-21 通过 CycleGAN 生成地图



图 8-22 通过 CycleGAN 将人物变为洋娃娃

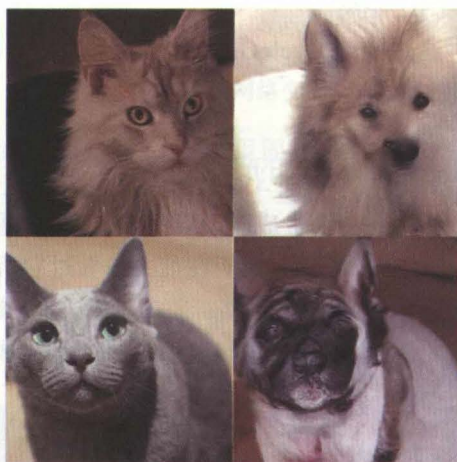


图 8-23 通过 CycleGAN 将猫变为狗

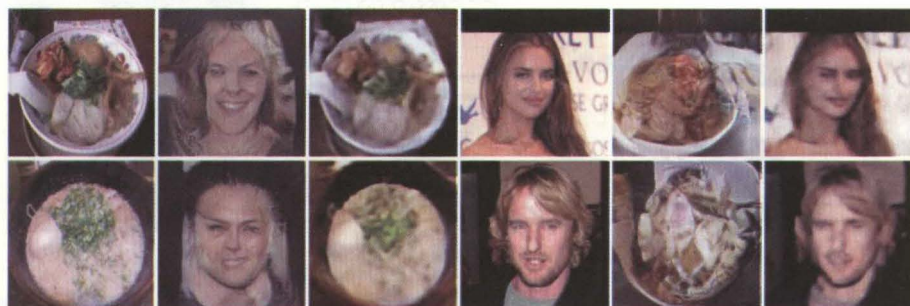


图 8-24 通过 CycleGAN 实现人和拉面之间的转换

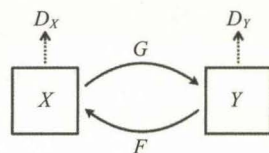
图 8-24 中的 6 列分别为：输入的拉面图像，变成的人图像，继续变回拉面图像（可见与此前的拉面图像相似），输入的人图像，变成的拉面图像，继续变回人图像（可见与此前的人图像相似）。

这就是 CycleGAN 的核心思想：将类别 X 的图像 x 变为类别 Y 的图像 y ，再变回类别 X 的图像 x' ，应与最初的图像 x 尽量相似。

给定 X 和 Y 两个数据集，构造下列网络，效果如图 8-25 所示。

- G 网络，用于将 x 转为 y 。
- F 网络，用于将 y 转为 x 。
- D_X 网络用于分辨 F 生成的 x 的真假。
- D_Y 网络用于分辨 G 生成的 y 的真假，

用 GAN 的方法训练 G , F , D_X , D_Y ，再要求 $F(G(x))$ 与 x 相似， $G(F(y))$ 与 y 相似，就完成了 CycleGAN 架构。具体方法是在 GAN 的损失函数中加入 $F(G(x))$ 与 x ， $G(F(y))$ 与 y 差异的损失项，可采用 L1 或 L2 距离。



最近 Facebook 的研究^①通过类似 CycleGAN 的方法，甚至可自动学会两种语言之间的翻译。

图 8-25 CycleGAN 的运作思想

8.4.2 生成高分辨率图像：nVidia 的改进

传统 GAN 方法的一大问题是难以生成高分辨率的图像，而 nVidia 在 2017 年 10 月提出的新训练方法^②很好地解决了这一问题，我们在前文已看过它所生成图像的惊人清晰度。

它的思路是逐步训练不断增大的 GAN 网络，首先实现 4×4 图像的生成和判别，然后实现 8×8 图像的生成和判别……最终可实现 1024×1024 图像的生成和判别，如图 8-26 所示。

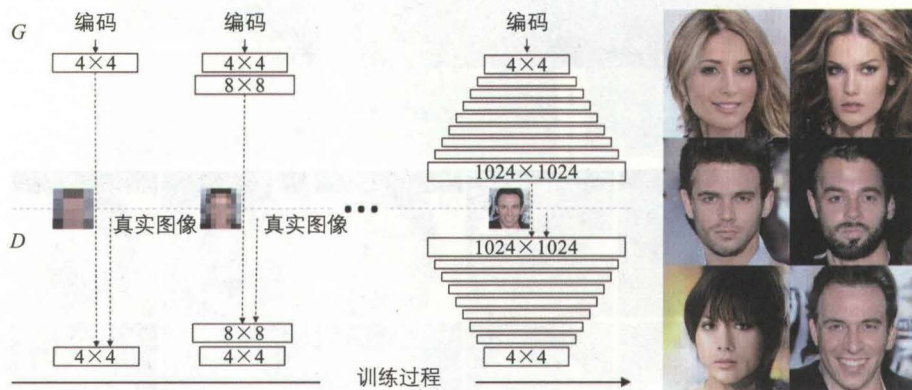


图 8-26 nVidia 提出的训练方法

这里的主要问题是如将将在 $N \times N$ 上训练好的网络继续在 $2N \times 2N$ 上训练。方法是使用类似残差的思想。图 8-27 中的 fromRGB 代表从 RGB 图像的映射，toRGB 代表向

① 地址为 <https://arxiv.org/pdf/1711.00043.pdf>。

② 地址为 http://research.nvidia.com/publication/2017-10_Progressive-Growing-of。

RGB 图像的映射。

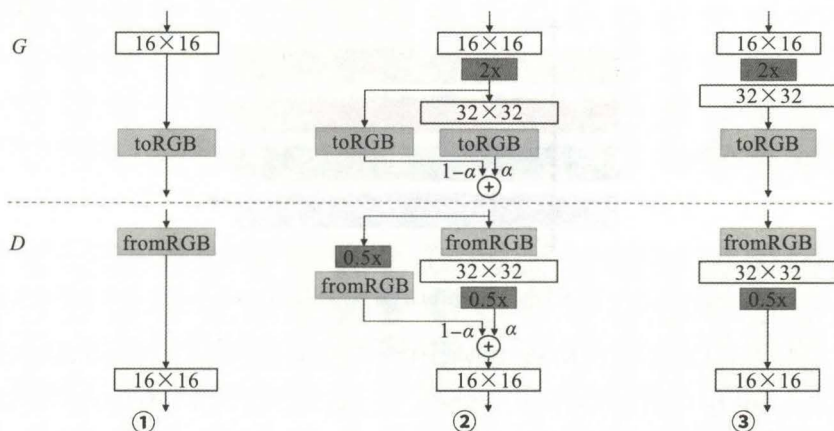


图 8-27 将 $N \times N$ 网络变为 $2N \times 2N$ 网络

1) 在 16×16 上训练 G 和 D 后, 先采用固定的放大和缩小方法 (如最近邻居放大, 2×2 平均池化缩小) 将它们变为可在 32×32 上运作。

2) 然后加入一个新的在 32×32 上运作的分支。令旧分支的权重为 $1-\alpha$, 新分支的权重为 α , 输出为它们的加权和。

3) 然后开始在 32×32 上训练, 最初将 α 设为 0, 即全部使用旧分支, 在训练过程中逐渐增大 α , 直到 α 等于 1, 此时就是完全采用新分支运作, 可移除旧分支。

8.4.3 自动提取信息: InfoGAN

在普通的 GAN 中, 编码 z 的每个维数不一定有明确的意义, 如图 8-28 所示。

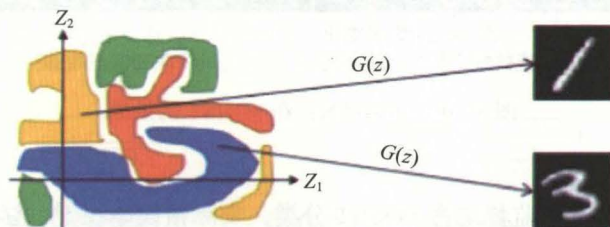


图 8-28 普通 GAN 中的编码

图中用不同彩色代表映射到的不同数字。虽然 G 能将编码 z 映射到 MNIST 中的数字图像, 但编码的 z_1 和 z_2 坐标并没有明确的意义。

为此, 研究人员提出了 InfoGAN[⊖], 它更为接近之前所述的“从图像自动总结出重要

[⊖] 地址为 <https://arxiv.org/pdf/1606.03657v1.pdf>。

因素”的梦想。InfoGAN 可实现如图 8-29 所示的更规整的编码空间。

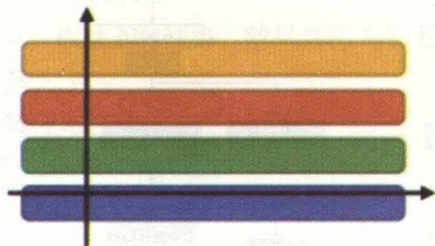


图 8-29 InfoGAN 中的编码

举例，如果使用无标签的 MNIST 数据集作为训练数据，然后要求 InfoGAN 总结出 3 个因素 c_1 、 c_2 、 c_3 ，其中 c_1 为离散的 10 个分类， c_2 、 c_3 均为 -1 到 1 的数，那么 InfoGAN 可自动总结出 c_1 对应数字的分类， c_2 、 c_3 分别对应数字的角度和笔画的粗细，如图 8-30 所示。

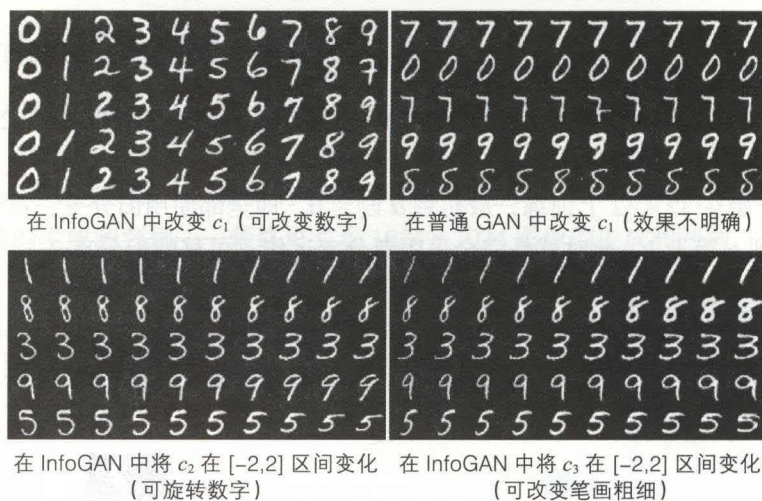


图 8-30 InfoGAN：在 MNIST 的效果

此图 8-30 说明：

- InfoGAN 几乎可无监督地将 MNIST 分类，实际错误率在 5% 左右（例如图中有个 7 被识别为了 9）。
- 在普通的 GAN 中改变某个编码，不一定能带来明确的语义效果。
- 在 InfoGAN 中，即使将 c_2 或 c_3 在更宽的 $[-2,2]$ 区间变化，也能得到高质量的效果。

在使用大量人脸模型图片训练 InfoGAN 后，它也能自动总结出其中最重要的因素，包括左右角度、上下角度、光照和脸的宽窄，如图 8-31 所示。

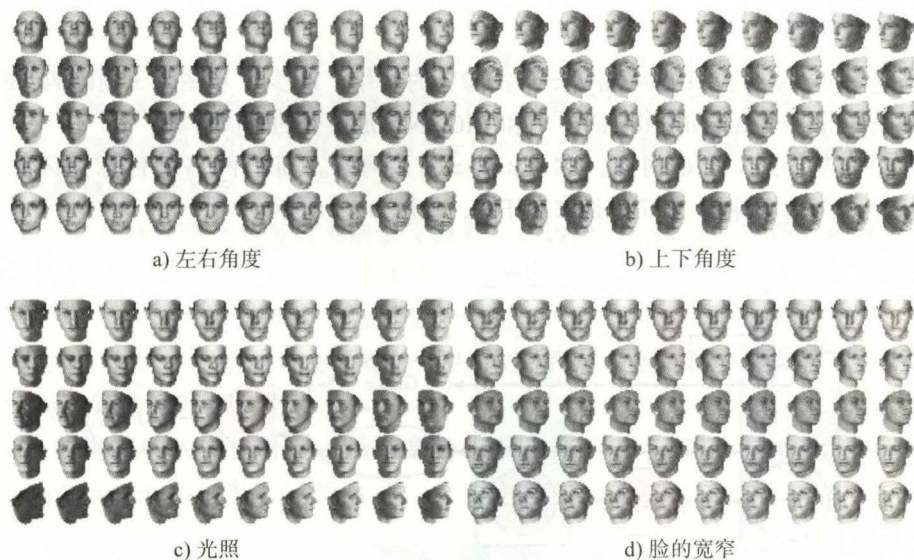


图 8-31 InfoGAN: 在人脸模型图片的效果

InfoGAN 方法的核心思想是:

- 将 G 的输入分为 c 和 z 两部分。其中 c 是符合所需格式的具有语义的隐变量 (latent variable), 而 z 是噪音 (即, 与图像有关, 但没有明确语义)。
- 要求 D 能同时输出对于真假的判别, 以及 c 。
 - 我们会在损失函数中加入 c 的还原误差的损失项。
 - 这会要求 G 必须在生成的图像中明确地运用 c , 这样 D 才有可能从图像中发现 c 。
 - 具体而言, 希望 D 能轻松还原出 c , 这相当于希望 G 很明确地运用 c 。我们会在 D 的倒数第二层加上分支 Q 输出 c 。于是这也类似于多任务学习。

可将普通 GAN 和 InfoGAN 的架构总结为如图 8-32 所示的架构。

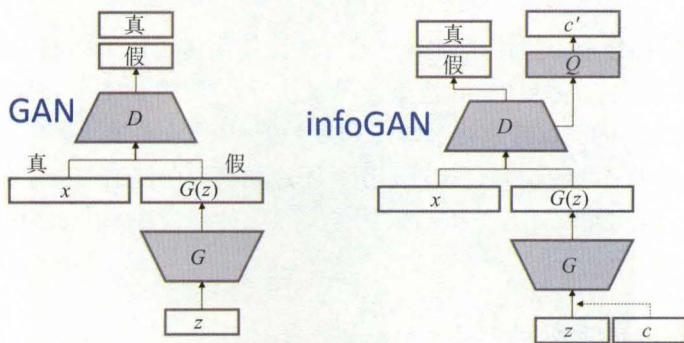


图 8-32 GAN 与 InfoGAN 的比较

8.4.4 更多应用

在普通的 GAN 中，只有从编码 z 到图像 x 的网络，缺少了从图像 x 到编码 z 的网络。ALI (Adversarially Learned Inference) ^① 和 BiGAN ^② 是漂亮的解决方案。

在其中有从 z 到 x 的 G 网络，和从 x 到 z 的 E 网络。而 D 网络的目标是区分二元组 $(G(z), z)$ 和 $(x, E(x))$ 是来自于 G 还是来自于 E ，如图 8-33 所示。

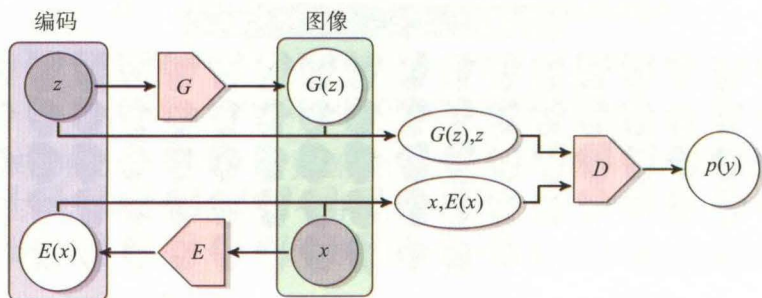


图 8-33 ALI 和 BiGAN 的原理

最终， G 可学会从 z 生成高质量的 x ，而 E 可学会从 x 还原出 z 。

通过 GAN 还可实现超分辨率 (super-resolution)，即将小图放大并保持清晰度。著名的例子是 SRGAN ^③，效果如图 8-34 所示。



图 8-34 通过 GAN 实现超分辨率

图 8-34 从左到右分别为：使用简单放大方法，使用传统的 SRResNet 网络放大（训练目标是让放大后的图像与原始图像在像素上尽量接近），使用 SRGAN 放大（训练目标是让 D 网络无法区分放大后的图像与原始图像），以及原图。

可见，SRGAN 的效果更清晰，接近原图，不过如果我们仔细观察，会发现它想象出了很多不同的细节，因此如果计算放大后的图像与原图的像素距离（如 MSE 距离），会发现 SRGAN 反而差别较大。

① 地址为 <https://ishmaelbelghazi.github.io/ALI/>。

② 地址为 <https://arxiv.org/pdf/1605.09782.pdf>。

③ 地址为 <https://arxiv.org/abs/1609.04802>。

这体现了 GAN 与传统方法的区别:

- ❑ 在传统方法中, 判别图像还原程度是使用像素还原误差, 但这会限制生成模型的运作。例如, 对于草地, 如果按照传统方法, 会要求生成模型去精确生成原图中的每根草的像素位置, 但这不可能, 因为 z 中的信息量不够。最终生成模型只能妥协, 得到模糊的一团绿色。
- ❑ GAN 中的 D 网络会更多地考虑语义因素, 如边缘的清晰度、纹理的细致度, 而不会纠结于噪音的细节。例如, 如果看上去像是草地, 那么就是合格的。具体每根草怎样排列并不重要, 重要的是能看到一根根草。

因此, GAN 中的 G 网络能生成更清晰、更真实的图像。但可能会包含与原图不同的细节, 例如草地中草的排列方法不同。

GAN 也能将缺失的图像补全, 目前最佳的效果来自 <http://hi.cs.waseda.ac.jp/~iizuka/projects/completion/en/>, 如图 8-35 所示。



图 8-35 通过 GAN 实现图像补全

最后, GAN 不仅能生成图像, 还可运用于诸多其他领域, 例如用于生成药物分子结构, 如图 8-36 所示。

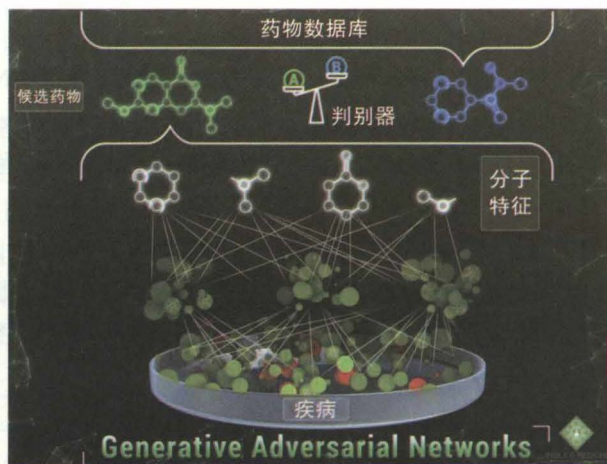


图 8-36 通过 GAN 研发药物

或是生成音乐[⊖]及视频[⊖]，甚至可用于生成密码以更有效率地破解账号 (PassGAN[⊕])。

8.5 更多的生成模型方法

除 GAN 外，目前还有其他采用深度网络的生成模型方法，而且它们的思想可互补，在此我们简单介绍。

8.5.1 自编码器：从 AE 到 VAE

自编码器 (Auto-Encoder, AE) 是经典的生成模型方法，其架构如图 8-37 所示。

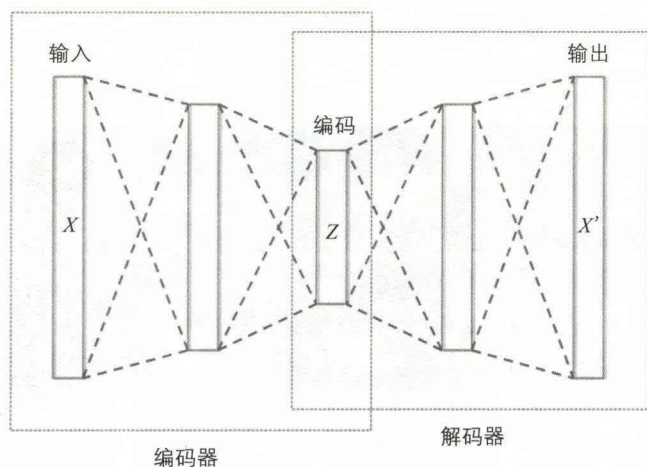


图 8-37 AE 的架构

AE 可分为两个网络：

- 编码 (encoder) 网络，负责从 x 到 z ，可称为 E 。
- 解码 (decoder) 网络，负责从 z 到 x ，可称为 G ，因为它和 GAN 中的生成网络都是从 z 到 x 。

可将从 x 到 z 看作压缩信息的过程，从 z 到 x 看作解压缩的过程。例如，如果 x 是 64×64 的彩色图像，那么它有 $3 \times 64 \times 64 = 12288$ 维。而 z 往往只有 50 到 200 维。

AE 和 GAN 的区别在于，AE 中没有更先进的判别网络 (D 网络)，AE 的优化目标只是让 x 和 $G(E(x))$ 尽量在像素上接近。如前文所述，这并不是个好目标，因此 AE 生成的图像往往很模糊，例如 AE 和 GAN 在 Fashion-MNIST 数据集的效果对比如图 8-38 所示。

⊖ 地址为 <https://arxiv.org/abs/1703.10847>。

⊖ 地址为 https://github.com/dyelax/Adversarial_Video_Generation。

⊕ 地址为 <https://arxiv.org/abs/1709.00440>。

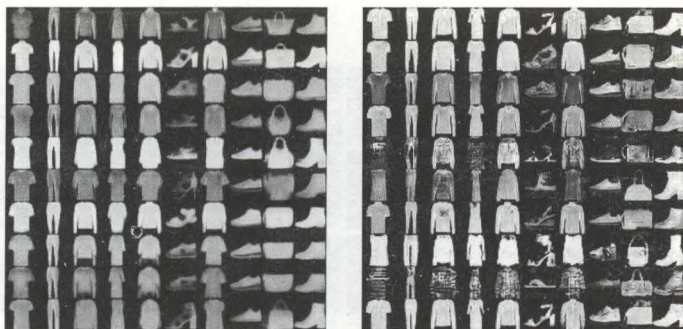


图 8-38 AE 与 GAN 的效果对比

不过, AE 相对于 GAN 也有长处, 就是生成的图像更均匀, 光滑。因此研究人员提出了多种将 AE 和 GAN 结合的方法, 后文会介绍其中之一。

AE 的重要发展是 VAE (Variational Auto-Encoder, 变分自编码器[⊖])。它能解决 AE 的一个缺点: AE 的 G 只能保证将由 x 生成的 z 还原为 x 。如果我们随机生成 1 个 z , 经过 AE 的 G 后往往不会得到有效的图像。

而 VAE 可让 E 生成的 z 尽量符合某个指定分布, 例如标准差为 1 的多维正态分布。那么此时只需从这个分布采样出 z , 就能通过 G 得到有效的图像。具体而言, 这是通过一个参数化技巧 (reparameterization trick) 实现, 可参阅 VAE 的原始论文。

举例, 对于 MNIST 数据集, 如果要求 z 是 2 维的, 最终效果如图 8-39 所示。

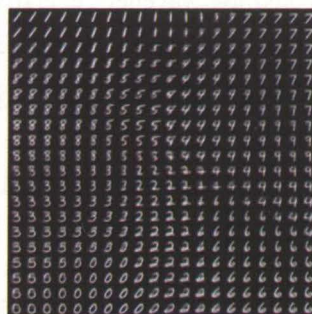


图 8-39 VAE 的编码空间

可见, 无论选取怎样的 z 坐标, 都能得到较为合理的数字图像。

8.5.2 逐点生成: PixelRNN 和 PixelCNN 系列

由 Google 提出的 PixelRNN[⊖]和 PixelCNN[⊗]是生成模型的另一种思路。

它的方法非常直接: 从左到右, 从上到下, 逐步生成一个个像素, 最终生成整张图像。如果读者熟悉循环神经网络 (RNN), 会意识到这是一个很适合 RNN 的问题。

基本原理如图 8-40 所示, 以之前生成的像素作为输入, 输出对于下一个像素值的统计分布的预测, 然后从分布采样出一个像素。

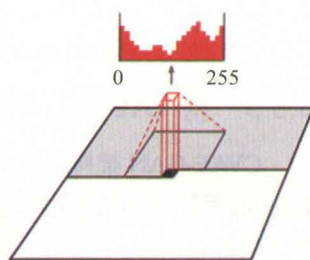


图 8-40 PixelRNN 的原理

⊖ 地址为 <https://arxiv.org/pdf/1312.6114>。

⊗ 地址为 <https://arxiv.org/abs/1601.06759>。

⊙ 地址为 <https://arxiv.org/abs/1606.05328>。

可以想象，它会很适合生成小图。例如图 8-41 中是它生成的珊瑚礁图像，色彩很鲜艳。



图 8-41 PixelRNN 的效果

而它的缺点无疑就是速度，以及目前仍然难以生成大图。于是读者可能会问，是否可构建出 PixelGAN？答案是肯定的^①。

最后，Pixel 系列的思想尤其适合生成音频和文字，例如 WaveNet (<https://deepmind.com/blog/wavenet-generative-model-raw-audio/>)，它用此前生成的音频采样作为输入，生成下一个采样，不断重复此过程，最终可生成高质量的语音和音乐，如图 8-42 所示。

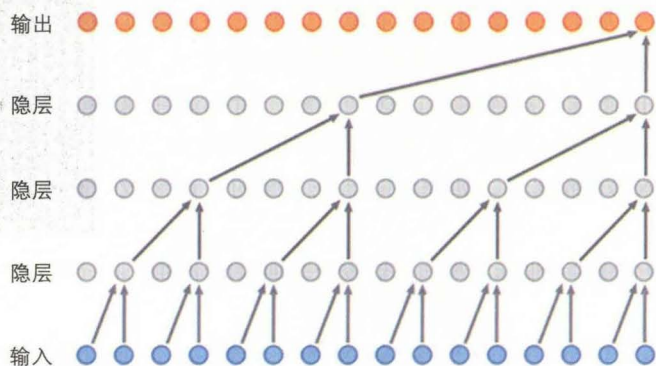


图 8-42 WaveNet 的原理

8.5.3 将 VAE 和 GAN 结合：CVAE-GAN

CVAE-GAN 架构的论文是《CVAE-GAN: Fine-Grained Image Generation through Asymmetric Training^②》，其中 C 代表能用分类作为输入，生成指定分类的图像。它在各个分类上生成的图像效果都相当好，如图 8-43 所示。

其中的图像都是由 CVAE-GAN 生成。它有 4 大组件，对应 4 个神经网络，互为补充，互相促进：

① 地址为 <https://arxiv.org/abs/1706.00531>。

② 地址为 <https://arxiv.org/pdf/1703.10155v1.pdf>。

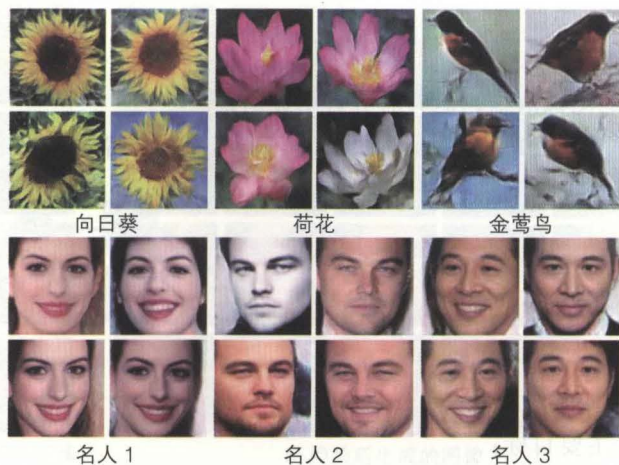


图 8-43

- E : 编码器 (Encoder), 输入图像 x , 输出编码 z 。如果还给定了类别 c , 那么生成的 z 就会质量更高, 即更随机, 因为可移除 c 中已包含的信息。
- G : 生成器 (Generator)。输入编码 z , 输出图像 x' 。如果还给定了类别 c , 那么就会生成属于类别 c 的图像。
- C : 分类器 (Classifier)。输入图像 x , 输出所属类别 c 。这是我们的老朋友。
- D : 判别器 (Discriminator)。输入图像 x , 判断它的真实度。

我们先看如果只使用部分组件会是怎样。首先是 CVAE, 如图 8-44 所示。

然后是 CGAN, 其中 y 代表对于真实度的判别, 如图 8-45 所示。

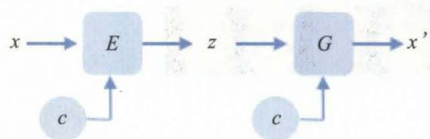


图 8-44 CVAE 架构

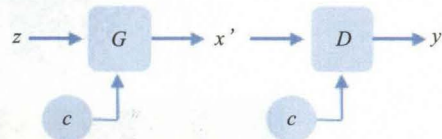


图 8-45 CGAN 架构

它们的效果如图 8-46 所示。



图 8-46 CVAE 和 CGAN 的效果对比

可见：

- CVAE 生成的图像中规中矩，但是模糊。
- CGAN 生成的图像清晰，但有时会有明显错误。

所以 AE 和 GAN 的方法刚好是互补。CVAE-GAN 的最终架构如图 8-47 所示。

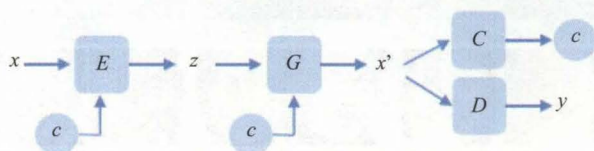


图 8-47 CVAE-GAN 架构

其中， G 有 3 个主要目标：

- 对于从 x 生成的 z ， G 应能还原出接近 x 的 x' （像素上的接近）。这来自 AE 的思想。
- G 生成的图像应可由 D 鉴别为属于真实图像。这来自 GAN 的思想。
- G 生成的图像应可由 C 鉴别为属于 c 类别。这与 InfoGAN 的思想有些相似。

最终得到的 z 可相当好地刻画图像。例如，同样的 z 在不同 c 下的效果如图 8-48 所示。

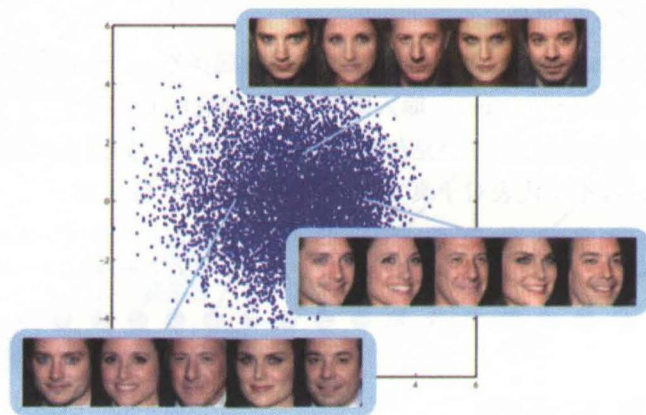


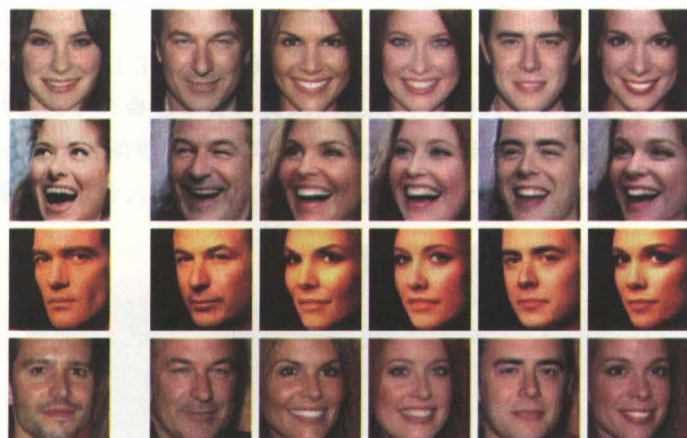
图 8-48 CVAE-GAN 的编码空间

这里的不同 c ，代表不同的明星。相同的 z ，代表其他的一切语义特征（如表情、角度、光照等）都一模一样。

于是，通过保持 z ，改变 c ，可轻松实现真实的换脸效果，如图 8-49 所示。

CVAE-GAN 在语义插值上的效果也很出色，如图 8-50 所示。

由于 CVAE-GAN 生成的样本质量很高，还可用于增强训练样本集，使其他模型（如图像分类网络）得到更好的效果。



a) 真实图像

b) 换脸后生成的图像

图 8-49 CVAE-GAN 实现的换脸



图 8-50 CVAE-GAN 实现的编码插值

通向智能之秘

我们在前文看到了深度神经网络的强大，但 AI 的综合能力离人类智能实际仍然有较大差距。在本章我们将讨论这些差距，它们的成因和可能的解决方法，并展望未来。

9.1 计算机视觉的难度

2016 年 5 月 7 日，一辆开启自动驾驶模式的 Tesla Model S 在美国发生严重车祸，全速撞上了一辆正在横穿高速公路的白色拖挂卡车的侧面，并从卡车下面钻了过去，Tesla 车主当场身亡。图 9-1 为事故中的白色拖挂卡车。

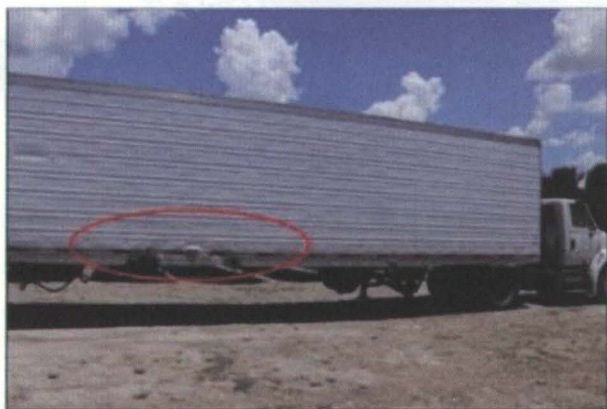


图 9-1 Tesla 自动驾驶事故中的另一方

在 Tesla 公司的调查报告中写到：

当时 Model S 行驶在一条双向、有中央隔离带的公路上，自动驾驶处于开启模式，此时一辆拖挂车以与 Model S 垂直的方向穿越公路。在强烈的日照条件下，驾驶员和自动驾驶都未能注意到拖挂车的白色车身，因此未能及时启动刹车系统。

Tesla Model S 的自动驾驶系统具有多个摄像头、超声波传感器、雷达，采用图像识别技术与多传感器信息融合，能在绝大多数情况下自动分辨出路面、车辆等，如图 9-2 所示。



图 9-2 Tesla 的自动驾驶系统

然而在特殊环境下，例如在面对相似的白色天空与白色车身时，摄像头的图像识别出现了致命的错误，这也说明了计算机视觉的难度。Tesla 公司在后续发布了自动驾驶 2.0 系统，增加了处理器的计算速度，配备了更多的摄像头与传感器。

在图像识别任务中，深度卷积网络也可能会出现人类不会犯下的错误。在著名研究者 Andrej Karpathy 的博客中，以 ImageNet 分类问题为例，对人类和当时电脑（当时电脑的错误率在 6.8% 左右）的优势和劣势有详尽的分析（<http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/>），如图 9-3 所示。

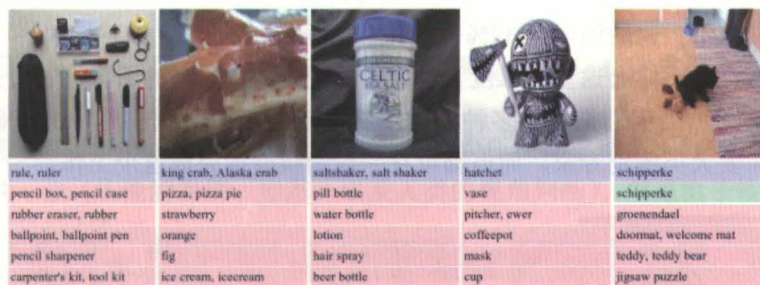


图 9-3 ImageNet 中的典型问题

- ❑ 第 1 张图的正确分类是尺子。电脑将其识别为铅笔盒。
 - 这并不算严重失误，因为图中的物体很多，并不容易说出哪个是正确答案。
 - 这可认为是 ImageNet 的设计缺陷，因为 ImageNet 的分类只允许 1 个正确答案。
- ❑ 第 2 张图是帝王蟹。电脑将其识别为披萨。
 - 这个错误可谓具有“想象力”，因为图中白和红的搭配有些像披萨饼。如果只让人看 0.1 秒钟就将图像撤走，人也有可能看错。

- 目前的深度卷积网络在图像识别出错时，经常会是这种“有想象力的错误”。
- 另一方面，这说明了电脑的思维比人类简单得多。目前深度卷积网络并未建立起物体的真正三维立体观念，而只是依照局部和全局的统计分布情况判断。
- 第3张图是盐瓶，因为上面写着大大的 SEA SALT 字样。电脑将其识别为药瓶。
 - 严格说这对电脑不公平，因为电脑在此只看过 ImageNet 的 120 万张图像，对其余常识一无所知。人类具有远远更多的经验。
 - 目前的图像识别深度卷积网络，尚不会去识别图中的文字做语义分析。但这里并没有技术上的障碍，完全可以配合文字识别网络解决这个问题。
 - 如果训练样本中频繁出现“SALT”代表盐瓶，电脑也会学会“SALT”与盐瓶具有关联性。
- 第4张图的正确分类是手斧。电脑将其识别为花瓶。
 - 这里玩偶的黑白色花纹并不是典型的绒毛玩具纹理，它可能让电脑想起了花瓶的图案。
 - 玩偶的直立形状略微有些像花瓶。两个因素综合，电脑做出了错误的判断。
 - 这里的正确分类也颇为隐蔽和不典型，手斧的纹理也与普通手斧不同。如果图中的物体是一把普通的手斧，电脑就几乎不可能出错。
- 第5张图显示了电脑的强项。电脑给出了正确分类：西帕基犬。
 - 在 ImageNet 的分类中有超过 120 种犬类。对于这种精细的分类问题，普通人并不擅长，电脑比普通人要强得多，达到了专家级的水平。
 - 在电脑的所有错误中，只有 7% 是这种精细分类问题的错误。而在 Andrej 自己的尝试中，他的错误有 37% 是精细分类问题的错误。
 - 我们可看图体会这里的难度。图 9-4 中左边是待分类的图像，右边是部分犬类的示例图。当我们把 120 多种犬类图看完后，是否可选出唯一的答案？这可能需要犬类鉴别专家。而电脑可在一瞬间很好地完成这个工作。



图 9-4 ImageNet 的难度

Andrej 自己在 ImageNet 上的分类错误率是 5.1%。他估计，如果由多位人类专家组队并仔细训练，最终可到达 2% 的错误率。感兴趣的读者可访问 <http://cs.stanford.edu/people/karpathy/ilsvrc/>，测试自己的图像分类能力。而目前在 ImageNet 上最强的深度卷积网络，

也已达到 2.3% 的错误率。深度学习的进展可谓一日千里，Andrej 也在博文的续篇中惊讶于电脑的进步之快。

那么，电脑是否在不久就能将缺陷一个个修正，在计算机视觉上实现对于人类的全面超越？Andrej 对于电脑在近期实现图像理解仍不乐观，因为现实中的图像远远更难。请考虑如图 9-5 所示的这张照片。



图 9-5 一张有趣的图像

这张照片很有趣。我们可通过训练让电脑学会大致识别哪些照片有趣，但电脑难以真正体会照片的趣味来自何处，因为这需要很强的理解能力与全面的常识：

- ❑ 电脑需要理解这是一群人在更衣室里，且图中有多面镜子，其中的倒影并非实物。
- ❑ 电脑需要从所有人物的着装，以及奥巴马的侧面图像，确认图中的主角是奥巴马。
- ❑ 电脑需要从左边人物的姿态和立体的高度，确认他是站在体重称上（尽管体重秤的白色与墙壁的白色很像）。
- ❑ 电脑需要识别出奥巴马的脚正在立体的环境中压在体重称上，而正在称重的人尚不知道这一点（从他的表情、姿态，以及人类的视角判断），但是由于镜子的反射，他可能很快就会发觉。
- ❑ 电脑需要理解这会让体重秤的读数变大，而这会让正在称重的人感到疑惑，因为人不希望自己的体重变大。电脑需要理解他可能会抬头观察，并发现原来是奥巴马在开玩笑。
- ❑ 电脑需要从所有人的表情明白这是在开玩笑。而且由于开玩笑者是奥巴马，这张图像就更为有趣。

我们可用硬性的规则手动让电脑学会以上知识，但这就不是机器学习的原则了。如何让电脑自动发现和运用以上知识？这确实仍是个艰巨的课题。

虽然目前深度网络尚未真正理解图像，但如果单论结果的准确率，深度网络又确实已在许多数据集上达到和超越人类。那么，应该如何理解这个看似矛盾的现象？让我们看下一节，考察深度网络的运作细节。

9.2 对抗样本，与深度网络的特点

早在 2013 年，研究者已发现，如果将图像加入少许精心构造的噪音，就能让深度网络的识别出错（<https://arxiv.org/abs/1412.6572>）。这种样本称为对抗样本（adversarial example），如图 9-6 所示。



图 9-6 对抗样本

左图可被深度网络正确识别为有 57.7% 的概率是熊猫，但在加入少量如中间所示的噪音后得到的右图，会被深度网络认为有高达 99.3% 的概率是长臂猿（尽管左图和右图在人类眼中毫无区别）。

更重要的是，这并非深度网络的复杂结构所造成的特殊缺陷，而是来自于神经元的线性输入特点。举例，假设神经元有 10 个输入，权重为：

$$w = [-1, -1, 1, -1, 1, -1, 1, 1, -1, 1]$$

输入为：

$$x = [2, -1, 3, -2, 2, 2, 1, -2, 5, 1]$$

那么将 w 与 x 逐项相乘，会得到 -1 。

现在我们将 x 稍做改变：

$$x = [1.5, -1.5, 3.5, -2.5, 2.5, 1.5, 1.5, -1.5, 4.5, 1.5]$$

每一项都只改变了 0.5，但是 w 与 x 逐项相乘就会是 4，改变的程度高达 5。这是因为所选取的改变方向恰好是最能让结果改变的方向。

根据最新研究^①，有时仅需变动 1 个像素就足以骗过深度网络，如图 9-7 所示。



图 9-7 仅用 1 个像素就可骗过深度网络

这里的第 1、2、4 张都会被错误识别为狗，而第 3 张会被错误识别为飞机。

① 地址为 <https://arxiv.org/abs/1710.08864>。

我们还可构造出在现实生活中也能骗过深度网络的物体[⊖]。这需要物体在缩放、旋转、模糊等之后仍然能保持欺骗性，演示视频为 <http://www.labsix.org/physical-objects-that-fool-neural-nets/>，其中的乌龟模型，无论从正面、侧面、背面各个角度观察，都会被深度网络识别为步枪，如图 9-8 所示。

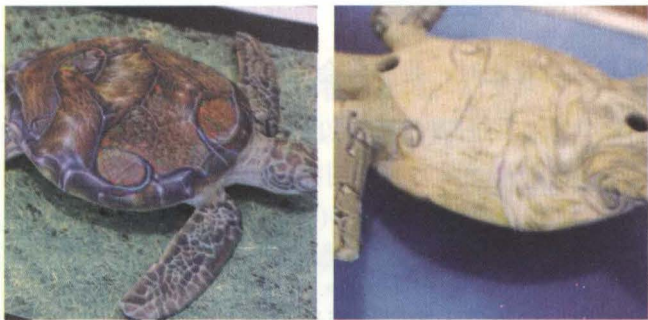


图 9-8 用实际物体骗过深度网络

在笔者看来，这涉及人类和深度网络进行视觉识别时的重要区别。当我们第一眼看到一张图像，我们的直觉与目前的深度网络有类似之处。但我们在仔细观察图像后，脑海中还会有立体、连续的场景重建，包括三维形体，纹理，光照，等等，再根据立体的场景进行后续的分析。

目前的深度网络，在处理图像时缺少明确的场景重建过程，属于纯粹的端对端学习，因此对于特殊场景会容易出现漏洞。

例如，人类很容易认出图 9-9 中的照片都是在描述同一座雕像，但深度网络需经过许多训练样本才能确认这一点。

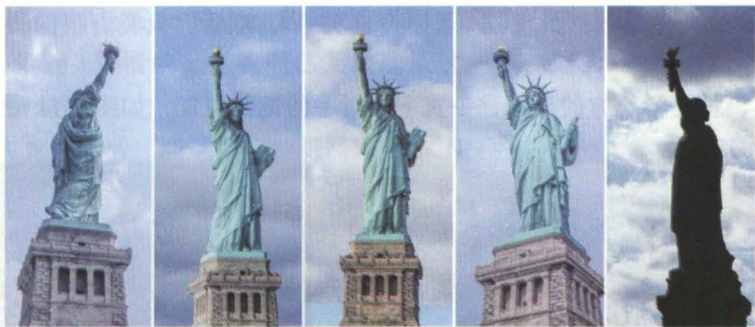


图 9-9 场景重建的难度

上述问题称为 Inverse Graphics，目前已有不少研究，如从人的面部照片构造出面部的立体模型。读者在 youtube 也可找到 Hinton 对此的看法——《Does the Brain do Inverse Graphics?》。

⊖ 地址为 <https://arxiv.org/pdf/1707.07397.pdf>。

另一方面，深度网络对于细节的敏感度比普通人类更高，对于种种统计规律的记忆和运用胜过人类，因此更擅长精细的分类。某种意义上，正是因为深度网络注重图像的细节，因此更容易被图像细节的改变所欺骗。

9.3 人工智能的挑战与机遇

在讨论人工智能与人类智能时，我们将会逐渐涉及种种复杂和未知的领域。本节属于随想形式，可供感兴趣的读者思考。

9.3.1 棋类游戏电脑陷阱

棋类游戏，是 AI 已取得诸多成功的领域。但即使在看似早已被电脑攻克的象棋中，也仍然存在对于 AI 困难，对于人脑简单的局面。典型的例子是“盲公顶棍”，如图 9-10 所示。

这个残局的奥秘在于，它等价于数学博弈论中的 Nim 游戏，正解是“华山一条路”，颇为隐蔽。一个聪明的人类有能力发现这一点，并运用数学知识得到完美的解答，而象棋 AI 却只会按部就班地搜索，很难发现最终的正解，因为象棋中的普通知识对于这个残局没有用处，这个残局已然是一种全新的游戏。

这体现了人类智能的隐蔽威力。我们有能力跳出原有的思维模式，从崭新的、更高的角度观察问题。这是非常奥妙和不可思议的能力。当牛顿从苹果落地看到万有引力，当爱因斯坦从电梯实验看到等效原理，其中所闪烁的正是这种能力的光芒。

- ❑ 这需要逻辑，因为需要进行真正的逻辑推理，而不是简单的博弈树搜索。
- ❑ 这需要常识，因为如果已经知道 Nim 游戏的存在，无疑会大大加快推理过程。
- ❑ 这需要创造力，因为一切的第一步，是体察到这个局面的特殊性。

而在围棋中同样存在对于电脑的陷阱，例如持白模仿棋，征子，摇橹劫和复杂劫争，细长大龙和乱战的大型死活，倒脱靴（通过自杀反杀），双活，人类有很深研究的某些特异角部死活，等等。

1) 模仿棋，就是学对方的下法，一直下在对手的镜像位置，如图 9-11 所示。

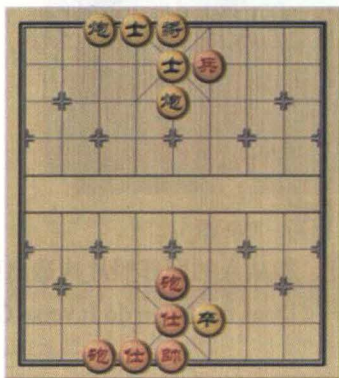


图 9-10 象棋中的“盲公顶棍”排局

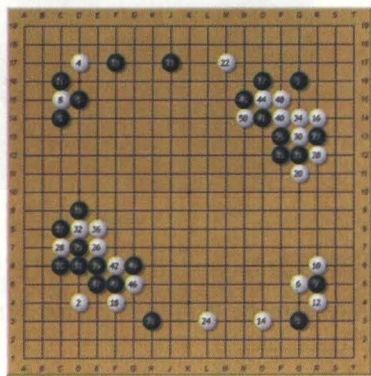


图 9-11 围棋中的模仿棋

模仿棋有多种破解方法。但由于 AI 无法自行意识到对手在下模仿棋（我们可通过人工加入判断教会电脑，但这不是机器学习的原则），因此会被人类使用策略克制。

2) 征子，是人类学习围棋时最早学会的概念之一，但对于 AI 却不简单，因为征子需要很深的直线计算，而且如果双方下错了一步就会形成完全相反的结果。在 AlphaGo Zero 的论文中也表示，AlphaGo Zero 通过自我对弈很快就学会了死活，厚薄等概念，但直到很久之后才逐渐学会征子。

尤其有趣的是，如果只让 AI 从人类高手棋谱学习，AI 容易对征子产生错误的看法，因为在人类高手的对弈中，如果出现了看似可以征子的局面，通常实际都不会去征子，因为对方可能已在远方做好了防御措施，称为“引征”（这是征子的另一个难点所在，它往往涉及远方的棋子，这对于深度卷积网络有难度）。但 AI 很难意识到这一点，只会简单地学会“高手下棋时不会去征子”，结果在与人类对弈时，也不去提防对手可能会征子，出现令人捧腹的失误。

目前在绝大多数围棋 AI 中，包括 AlphaGo Master，都需要人工加入对于征子的特别处理。虽然 AlphaGo Zero 看似不再需要这种人工处理，但我们并不知道它是否仍会在罕见的情况下出现失误。从 Leela Zero 的情况看，AlphaGo Zero 很可能只是将漏洞藏得更深。

与前文的象棋局面一样，征子可认为是围棋中的一种自成体系的小游戏。对于这种游戏之上的游戏的认识和运用，是人类智能的强项。

3) 类似的例子还包括打劫，例如，摇橹劫是一种较为罕见的棋形，可为一方提供无限的“劫材”，如图 9-12 所示。

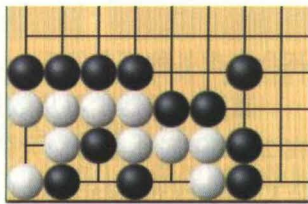


图 9-12 围棋中的摇橹劫

但 AI 很难发现这一点，出现这种局面时往往会输给人类。例如，日本的 DeepZen，腾讯的绝艺，中国台湾的 CGI，甚至在 AlphaGo Zero 和 AlphaGo Master 的自战对局中也隐隐有仍然不识摇橹劫的痕迹（可惜的是，我们已无法确认这一点，因为 AlphaGo 已经引退）。

4) 细长大龙的死活，涉及深度卷积网络（DCNN）的另一缺陷，即 DCNN 无法建立明确的拓扑概念。举例，当人类看到一张包括了一个圆的图像，即可提炼出连续的“圆”概念，无论它如何变形，都能识别出圆。但 DCNN 无法保证做到这一点，它只能通过使用大量的训练数据和数据增强，趋近于做到这一点。而且 DCNN 在一层层向上传递信息的过程中，有可能丢失各种信息：

- 如果一条很长的大龙只在一端有两个真眼，另一端就不一定知道自己已经活了。
- 如果大龙的气很多，DCNN 就有可能倾向于认为它活了，尽管它不一定有眼位。

通过让 DCNN 学习更多具有很长的大龙的局面，以及配合更深的残差架构，可改善这一点，但 DCNN 确实无法保证学会这个概念。

不过，虽然 AI 仍然存在种种缺陷，但人类很难引出这些缺陷，因为 AI 在大局观，局

势评估等方面已遥遥领先人类，而且价值网络会让 AI 避免进入其难以掌握的局面（所以我们无法知道，AlphaGo 是解决了这些缺陷，还是学会了有效地隐藏这些缺陷）。如果人类和 AI 正面比拼内力，而不是精心“设陷阱”“找 bug”，会极难取胜。

在笔者写作本书时，多个实力极强的 AI 正在网上不断与人类顶尖棋手对弈，过程很有趣：

- 有的棋手属于力战型棋风，和 AI 硬对硬，结果往往会被完全碾压，只见 AI 妙手迭出，人类疲于奔命，难有招架之力，最终只会连输多盘，被“抬走，换下一个”。
- 有的棋手选择运用种种“反 AI”战术，在棋盘各处理下地雷，有时可取得奇效，令 AI 晕头转向，棋力一落千丈，观战者纷纷表示“狗狗又发疯了”（由于大家将 AlphaGo 俗称为“阿尔法狗”，因此目前所有围棋 AI 的绰号都是“狗”）。

如果看 AI 赢的棋，你会觉得 AI 无懈可击，和人类已经不在一个层次。

如果看 AI 输的棋，你又会发现它输的方法很可笑，会往人设好的圈套里面钻。不过，现在也只有顶尖棋手才有可能成功给 AI 下套，而且不能保证成功率。

看来，如果人类在未来需要与 AI 对抗，AI 的“智慧”“大局观”恐怕会比人类更强，我们所能依靠的就是人类的“狡猾”“不按常理出牌”。

在某些方面与人类接近，在某些方面特别强，在某些方面特别弱；强大的时候宛如神灵，但如果被抓住 bug 则又会快速崩溃；这种种矛盾的统一，是目前 AI 的典型特点。

9.3.2 偏见、过滤气泡与道德困境

在上文我们讨论了目前的深度网络和 AI 在技术上的一些缺陷。而 AI 在实际中的应用，还会出现各种复杂的不一定属于技术性的问题。

首先，机器学习从数据中提炼规律时，有可能会得到与社会的道德规范相违背的结论。在 2016 年底，上海交大的研究者发布了一篇在全球范围引起广泛争论的论文《Responses to Critiques on Machine Learning of Criminality Perceptions》^①，其中的深度卷积网络，可以从类似图 9-13 中的照片识别出照片中的人物是否是罪犯，准确率达到 90%。

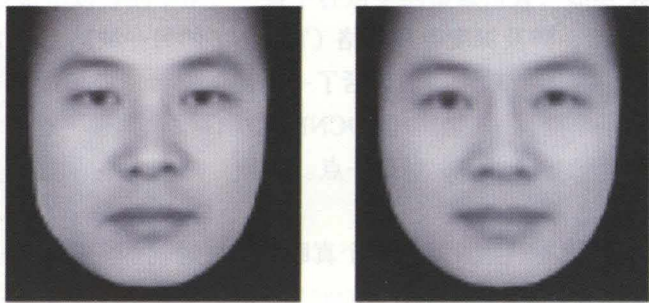


图 9-13 “以貌取人”的深度网络

^① 地址为 (<https://arxiv.org/abs/1611.04135>)。

上图的左边是论文中发现的“罪犯平均样貌”，右边是“非罪犯平均样貌”。这令人想起好莱坞电影《少数派报告》和美剧《疑犯追踪》。如果未来广泛运用类似的技术，对社会将产生怎样的影响，值得深思。

这篇论文在发布后立刻遭到了大量抨击。另一方面，我们恐怕在暗地里会同意，右边的样貌看上去相对更和蔼可亲，更面善。虽然是否面善，和是否会犯罪，没有直接联系，但确实会互为影响。这正如我们无法否认样貌的美丑在客观上对于人的发展存在显著影响，尽管每个人都会说自己不会以貌取人。

因此，许多研究人员认为，AI的发展，需要人为加入“反歧视”“反偏见”的模块，以防止它导致社会中歧视和偏见的恶化与恶性循环。

类似极具争议性的研究，还包括 Stanford 大学研究人员发布的可识别同性恋的深度卷积网络 (<http://goo.gl/BvAh9g>)。图 9-14 是它发现的男女同性恋和同性恋的平均样貌。

它仅需 1 张照片，对于男女性取向的判断准确率分别达到 81% 和 71%。而人类通过 1 张照片对于男女性取向的判断准确率只有 61% 和 54%。这篇论文也引发了诸多媒体和评论人士的强烈谴责和极大愤慨。耐人寻味的是，看上去它生成的平均样貌也似乎有一定道理。

此外，深度网络还有可能形成性别歧视。例如，研究人员发现：

- 如果让电脑回答“男人：程序员 = 女人：？”的问题，答案会是家庭主妇。
- 深度网络会自行发现“厨房”与“女性”的相关性，甚至可能将 1 张图像中身处厨房的男性判定为女性。

如何防止机器学习方法出现诸如此类的偏见，是目前的研究话题之一。

机器学习带来的另一问题是过滤气泡 (filter bubble)，如图 9-15 所示。如果读者用过“今日头条”“网易新闻”等 App，会注意到它们会不断推送与刚刚点击过的新闻类似的新闻。

例如，如果读者点开了一则某支球队的体育新闻，就会不断看到同一支球队的新闻；如果点开了一则某位明星的小道消息，就会不断看到同一位明星的娱乐资讯。如《The Filter Bubble》一书所述，类似的现象在互联网上已成为常态。

在越来越多的情况下，我们在互联网上看到的都是根据我们的个人喜好、点击习惯、输入内容而精心调整的结果。归根结底，这是因为互联网公司希望尽可能地提高用户的点击率，



图 9-14 “以貌取人”的深度网络：另一例子

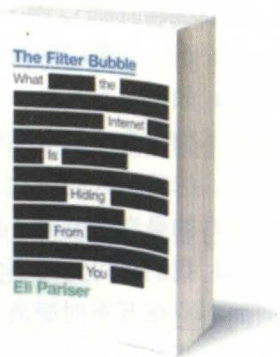


图 9-15 过滤气泡

提高用户的停留时间，提高广告收益，提高商品购买率，提高利润。机器学习的方法在此十分有效。互联网公司没有动机去向用户展示世界的全貌，而且，许多用户并不想看见世界的全貌，许多人只想看到自己想要看到的东西。

但，这是否正确？关心社会的读者可能已经留意到，最近几年在西方世界出现了越来越严重的社会分裂。而搜索引擎和社交媒体只会对此推波助澜，加剧分裂。例如，网站早已有能力判断出使用者是特朗普（美国共和党）还是希拉里（美国民主党）的支持者，从而只向使用者展示他所支持的一方的正面新闻，与他所厌弃的一方的负面新闻。

最近《人民日报》对此发表了评论：

毫无疑问，信息的私人订制能满足人们多元化、个性化的需求。智能化的信息传播机制可以快速完成用户与信息的精确匹配，大大降低获取信息成本，为生活带来便利。但换个角度看，算法主导下的内容分发模式，也会带来“自我封闭”的危险。

传播学有种回音室效应，在算法的帮助下，我们可以轻易过滤掉自己不熟悉、不认同的信息，只看我们想看的，只听我们想听的，最终在不断重复和自我证成中强化了固有偏见和喜好。一旦身处这样的“信息茧房”，就再难接受异质化的信息和不同的观点，甚至在不同群体、代际间竖起阻碍沟通的高墙。

机器学习可以对社会带来正面影响，也可以带来负面影响，这一切与优化的目标有关：

- 如果目标是最大化读者的点击率与停留时间，最终的效果就是过滤气泡。
- 如果目标是让读者尽可能获得更高质量、更全面的信息，也许会更有利于社会。但这不一定对公司的利润有利，而且读者也不一定喜爱这样的信息。

笔者认为，在此的解决方案，是让用户可自行决定是否关闭这种“智能推送”。令人遗憾的是，各大公司均未提供这种选项。它们的服务往往是免费的，因此用户没有选择的的话语权。所以说，免费的事情也是昂贵的。

最后，随着 AI 在现实中的更多应用，我们还会遇到种种道德与法律的困境。例如，在医疗领域，如果 AI 做出错误的诊断，责任该如何分配？

更复杂的例子，是自动驾驶：

- 当自动驾驶的汽车出现事故时，责任该如何分配？
- 如果汽车失灵，在“撞上侧面的树”与“撞上前方的行人”之间，汽车该如何抉择？如果自动驾驶系统推断出，撞上侧面的树意味着驾驶员的死亡，撞上前方的行人意味着行人的死亡，那么该如何抉择？
- 如果汽车失灵，必须选择撞上前方的多个行人中的一个，汽车该选择谁作为受害者？
- 我们还可构建更实际的法律问题，考虑这个问题：在一个大雾的天气里，驾驶员保罗在开车时睡着了，自动驾驶的汽车试图通过警报唤醒他，但他睡得很沉，于是汽车开启了警示灯并停到了路边。此时另一名驾驶员朱莉的车和它追尾，于是大家都很气愤，请问各方的责任该如何分配？很可能出现的情况是，双方选择共同起诉自

动驾驶汽车的厂商。于是汽车厂商必须将这种开销提前记入自动驾驶汽车的成本中。

由于自动驾驶无疑会在不久的将来成为现实，目前美国、德国等都已制订相应的法律。这会是人类历史上首次正式思考如何规范 AI 在社会中的运作。

9.3.3 语言的迷局

从基因的角度而言，人类与黑猩猩的相似度高达 98.7%，而根据研究人员的测试，黑猩猩的智力可达到人类的 3~10 岁水平，甚至在某些领域超过人类。例如，研究人员惊讶地发现，黑猩猩在数字记忆游戏中的速度比人类记忆冠军快 3 倍，如图 9-16 所示。



图 9-16 正在玩数字记忆游戏的黑猩猩

那么，为何人类在最近的几万年中发展越来越快，创造出了辉煌的文明，而黑猩猩却一直在原地踏步？这说明可能某些事物是关键，一旦突破，后续的发展就是一日千里。

在笔者看来，这个关键是语言。语言，可以有效地储存和传递高层次的知识。正如在围棋中，人类棋手通过学习棋书和聆听老师教授的棋理，在几千局对局之后，就能实现相当强的棋力。与人类棋手相比，AlphaGo 的学习过程需要远远更多的对局数，重要原因是 AlphaGo 无法掌握和运用语言。

语言也是逻辑和常识的集中体现，这两点正是目前 AI 的弱点。因此，如果 AI 能真正学会语言，将带来 AI 能力的重大飞跃。具体而言，可分为两个层面：

- AI 对于人类语言的理解。这称为 NLP (Natural Language Processing, 自然语言处理)。
- AI 创造出适合自己的语言。

先看理解语言。NLP 在历史上也曾有逻辑学派与统计学派之争，而目前基于深度学习的统计学派已占据优势。深度学习在翻译问题上尤其出色，性能远超此前的方法，接近人类水准，因为可用大量文本训练出准确的模型（例如使用 attention 机制的 LSTM 循环神经网络）。

但，这并不代表 AI 已理解人类语言。如果读者用过各家的语音助手，就会发现它们更像是“人工智障”：

- 当我们的问题是简单的查询类问题，如“明天的天气如何？”“苹果公司的股价是多少？”，目前的语音助手可给出准确的回答。
- 但如果我们的问题具有一定的逻辑推理关系，或是存在上下文的联系，或是包含常识，电脑所能做的往往就只是展示在搜索引擎搜索这个问题的结果。
 - 搜索的结果或许会让提问者满意，但这里没有任何智能，与我们手工把这个问题敲入搜索引擎没有任何区别。
 - 例如，“硬币和珠穆朗玛峰哪个更大？”对于 AI 非常困难。

根据《Intelligence Quotient and Intelligence Grade of Artificial Intelligence》^①的测试，目前“最聪明”的语音助手来自 Google，IQ 达到 47，相当于 5 岁小孩。而苹果的 Siri 的 IQ 只有 24。笔者认为，AI 的 IQ 在目前只有参考价值，因为 AI 在某些问题上特别强，在某些问题上特别弱，和人类智能的能力分布完全不同。

目前研究人员认为，NLP 中的问题，从易到难，顺序如下：

- ❑ 难度 1：文本搜索。
- ❑ 难度 2：文本分类、情感分析。
- ❑ 难度 3：翻译。到这里为止，AI 都已做到接近人类水平。
- ❑ 难度 4：文本摘要。这对于 AI 有一定难度。
- ❑ 难度 5：垂直领域问答。
- ❑ 难度 6：泛领域问答。语音助手正属于这一类。

这与 CV 的情况有些类似，容易的问题（如图像分类）已经被基本解决了，剩下的都是硬骨头，需要一步步慢慢解决。

读者可能会问，图灵测试是否也是关于理解语言的测试？并非如此，因为有许多技巧可骗过实验者（例如加入一些笑话、段子等），让实验者错以为自己在和真实的人类聊天。早在 2014 年就已有的程序成功通过图灵测试，但它并无智能可言。

研究人员在此的一个真正的努力方向，是希望 AI 首先能通过大学的入学考试。这仍然不意味着 AI 理解了试题，不过至少是一个更难以作弊的目标。例如日本的 Torobo-kun、美国的 GeoS 和 Aristo，它们在 2016 年都可拿到约 50% 到 60% 的分数。

那么，最新进展如何？遗憾的是，并不如人意。因为这些答题系统都是通过传统的逻辑推理方法，深度学习在此更容易显现其逻辑缺陷，而且难以收集到足够深度学习发挥威力的海量数据。

而传统方法是有致命伤的，历史已告诉我们：难以自动维护。Torobo-kun 团队已经放弃了继续开发。举例，尽管 Torobo-kun 的数据库里存储着曹丕是曹操的儿子，但它无法回答下列问题：“谁是曹丕的父亲？谁成为了中国三国时代魏国的第一位皇帝？”，因为它无法想到曹操就是曹丕的父亲，因为它不理解父子关系。毫无疑问，我们可通过加补丁的方法让 AI 学会这一点，但补丁要加到何时？AI 何时才能学会有效地总结和运用逻辑规律？

在笔者看来，让深度神经网络发现和掌握简单的逻辑规律，是可实现的目标（也许会来自 attention 与记忆机制的进一步发展），并且我们可能会在近年内看到这一点（前提是，有足够多的 AI 研究人员知难而上，因为目前的大多数研究人员更倾向于去解决易于解决的问题）。

这里困难的问题，是真正理解概念。姑且不提“意识”“时间”这些抽象概念，即使是

① 地址为 <https://arxiv.org/abs/1709.10242>。

“苹果”“猫”这样明显的具体事物，也存在着极其深奥的内涵，包含各个学科的知识，从形体和感官特征，到物理和生物属性，再到历史文化的意义，等等，一言难尽。

通过使用统计方法，目前的 AI 能够发现概念间的线性类比关系。例如，通过词嵌入 (word embedding)，AI 可自动发现如下的关系：

国王 - 女王 = 男 - 女
伦敦 - 英国 = 北京 - 中国

但目前的 AI 无法通过阅读自动建立对于概念的全面且精确的理解。

笔者认为，这里相对容易的，可能是从理解数学概念开始。某种意义上，数学是一门特殊的语言，而且它的规则较为明确。虽然“质数”“实数”等概念仍然有复杂的内涵，但至少我们已经掌握了精确定义和描述它们的方法，也已经有了初步的自动推理系统（如 Coq 和 Isabelle/HOL）。

笔者的预测是，深度学习和 AlphaGo 的方法，在此可能会奏效，我们有可能像捕捉棋感一样捕捉到数学灵感。这里的关键是，灵感不需要是完全正确的，只要它们能作为启发性的原则 (heuristic)，辅助推理过程，就足够了。数学证明的正确性，最终仍然可由推理过程保证。因此，这里的情况的确和 AlphaGo 有相似之处。

除此之外，程序语言也是相对简单的语言，目前已有不少“让 AI 学会自动编程”的研究（如微软的 DeepCoder），但它们离实用还有遥远的距离。

我们已经看到，理解语言困难重重，那么，创造语言是否更难？不一定，因为人类的语言形式不一定适合 AI，AI 可以逐渐发现更适合自己的语言。

在 2017 年 6 月，多家媒体报道，Facebook 的研究人员实验让 AI 聊天机器人学会互相讨价还价，发现 AI 在此过程中逐渐抛弃了人类语言的语法，创造出了自己的语言，而且 AI 可以此成功互相交易。

```
Bob: i can i i everything else . . . . .
Alice: balls have zero to me to me to me to me to me to me to me to me to me
Bob: you i everything else . . . . .
Alice: balls have a ball to me to me to me to me to me to me to me to me to me
Bob: i i can i i i everything else . . . . .
Alice: balls have a ball to me to me to me to me to me to me to me to me to me
Bob: i . . . . .
```

不过，这看上去更像是对信息做了简单的编码和解码，并没有实现人类语言的概括和抽象能力。

更进一步的研究来自 OpenAI (<https://arxiv.org/abs/1703.04908>)，研究人员构建了一个虚拟的二维环境，布置了一些 AI 智能体 (agent)，通过强化学习方法，希望让它们学会共同完成一系列任务，并允许它们使用语言交流。

例如，如果一个智能体发现在某个情况下向另一个智能体发送某个信息可以最终获得更高的奖励，就会学会在这个情况下运用这个信息。图 9-17 中显示了它们之间的语言的片段。



图 9-17 智能体发明的语言

研究人员发现，在只有 1 个智能体时，它不会去使用语言；在有 2 个智能体时，它们发明了 1 个词用于交流；在有 3 个智能体时，它们就学会了运用包含多个词的句子。

同时，智能体还学会了通过非语言的方法交流，如“通过望向目标，指示目标的位置”“在旁边跟着走”“在后面推着走”，如图 9-18 所示。

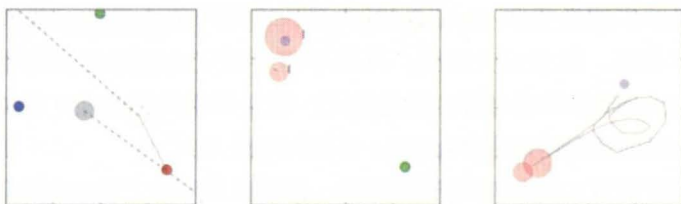


图 9-18 智能体用非语言的方法交流

这说明，AI 已有能力创造出适合自己的简单语言，难点在于如何提高语言的抽象能力。目前看来，在环境中进行强化学习，很适合于实验和培养 AI 完成这一目标。

9.3.4 强化学习、机器人与目标函数

强化学习是 AlphaGo 超越人类的关键，对于 AI 的自我进化同样至关重要。在许多强人工智能的路线图中（如 <https://www.goodai.com/>），强化学习，即对于环境的自动学习和适应，都是其中的中心话题。

Google DeepMind 在强化学习上投入了大量研究资源，例如，AI 早已能无师自通学会许多简单的 Atari 游戏，并能在大多数游戏上玩得比人类更好（<https://deepmind.com/blog/deep-reinforcement-learning/>）。

AI 也可在虚拟环境中学会有效地控制人形机器人，找到最佳的运动姿势（<https://deepmind.com/blog/producing-flexible-behaviours-simulated-environments/>），如图 9-19 所示，它发现浮夸的动作可让自己跑得更快。

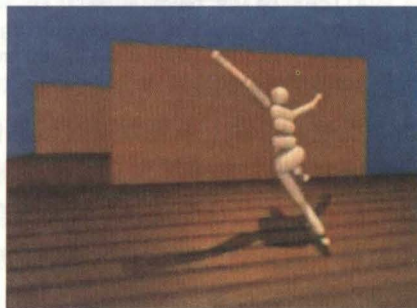


图 9-19 AI 发现的奇特跑步姿势

而 2017 年的另一则新闻是由 OpenAI 开发的 AI 在《Dota 2》游戏的 1 对 1 单挑中战胜了多位世界顶尖选手^①，如图 9-20 所示。

① 地址为 <https://blog.openai.com/dota-2/>。



图 9-20 正在玩《Dota2》的 AI

这个 AI 与 AlphaGo 一样，是从自我对战中学习。它的学习过程很有趣：

- 最开始，AI 只会躲在基地里，因为它发现离开基地后就会被对手击败。所以最佳的策略是躲得越远越好。
- 后来它经过不断尝试，逐渐学会了跟随部队与使用技能，最终可极其精准地运用各种策略，甚至有能力发现全新的罕为人知的战术。

与 AI 对战的多名职业选手在赛后表示，这个 AI 已不可能战胜。不过，当 OpenAI 将 AI 放到服务器上公开测试时，玩家很快发现可通过几种“流氓”战术将它击败^①。人类的这种“狡猾地找 bug”能力确实很有趣，是目前的 AI 难以学会的。

在更为复杂的游戏中，目前的 AI 也与人类存在差距。DeepMind 在 2016 年曾宣布在围棋后的下一个目标是让 AI 无师自通学会《星际争霸 2》游戏（著名的即时战略游戏），但 DeepMind 最终在 2017 年宣布中止研发，因为难度确实远远更大。最终 DeepMind 只公布了一个半成品（由屏幕图像识别出画面中的单位、建筑的状态），并表示期待有其他团队继续这个工作，如图 9-21 所示。



图 9-21 深度网络对于《星际争霸 2》画面的分析

① 地址为 <https://www.theflyingcourier.com/2017/9/11/16285390/elon-musk-open-ai-esports-bot-dota-2-defeated-beaten>。

《星际争霸2》的难点在于，其中的许多决策在很久之后才能显示效果。例如，如果花费一些资源建立一个新基地，需要等待基地建设完成，并调集采集单位，才能开始收回此前的建筑成本。

在笔者看来，这可能对 AI 有些苛刻。因为人类在学习玩游戏时，会先了解游戏的玩法，例如阅读游戏指南，而不是从零开始摸索。举例，如果要求一位从来没有接触过电脑游戏，而且不懂英文的人类，学会如何玩一款复杂的英文游戏，恐怕也会是个艰难和漫长的过程。目前已有研究者试图让 AI 理解游戏规则的文字。

如果我们能让 AI 在虚拟游戏中通过强化学习实现自我进步，那么下一步就是将这种能力带入现实世界。机器人技术在此会变得更为重要：如果 AI 能通过机器的躯体，主动探索世界，那么它的学习过程就会更为迅速，更有可能解决现实中的问题。

但 AI 与机器人的结合，也可能给人类带来更大的威胁。例如，Google 在 2013 年收购知名机器人公司 Boston Dynamics，但最终在 2017 年将其出售，重要原因是，这些机器人很可能会被用于军队，对于公司形象产生负面影响，如图 9-22 所示。

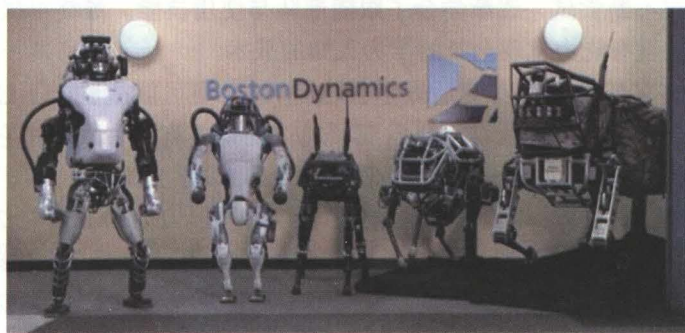


图 9-22 Boston Dynamics 的机器人

AI 与无人机的结合还能制造出可怕的空中杀手，美军已多次使用这一手段。这一技术可进一步小型化，一个鸡蛋大小的微型无人机就已能装载足够的炸药对人造成致命伤害，并能配合人脸识别，进行精准攻击。

在 2015 年和 2017 年，包括霍金、Elon Musk 在内的诸多知名人士签署了请愿书，要求禁止制造使用 AI 控制武器。联合国为此召开了“杀手机器人”会议，但最终没有产生任何决定。

目前看来，AI 在军事上的应用是不可避免的，因为它可带来武器威力的巨大提升，正如 2016 年美军的推演 (http://magazine.uc.edu/editors_picks/recent_features/alpha.html) 所显示的，如图 9-23 所示。



图 9-23 AI 与人类顶尖飞行员的对战

图中红色飞机是 AI 操纵的编队，通过灵活的引诱和躲闪，以及完美的配合，在多次实验中 AI 均保证 100% 击落所有蓝色的由人类顶尖飞行员操纵的飞机，且自己毫发无损。人类在尝试多种战术后仍然一筹莫展。不过，这里的 AI 是由人类飞行员调校出的，输入了大量人类战术，并非纯粹的自我学习。

让我们将目光移回 AI 理论。早在本世纪初，研究人员已运用强化学习观点，提出了通用人工智能 (Artificial General Intelligence, AGI) 的公式，称为 AIXI：

$$\arg \max_{a_t} \sum_{o_t} \cdots \max_{a_m} \sum_{o_m} [r_t + \cdots + r_m] \sum_{q: U(q, a_1 \cdots a_m) = o_1 r_1 \cdots o_m r_m} 2^{-\text{length}(q)}$$

其中 t 代表时间， m 代表智能体的生命期， a 代表行动， r 代表奖励， o 代表观测， U 代表通用图灵机， q 代表所有图灵机程序。整个式子的意思，是通过图灵机对于过去的行动、观测、奖励建模，从而找到能最大化未来的奖励的行动。

但这里仍然有很多问题：

- AIXI 只拥有理论意义，因为图灵机的停机问题已是不可计算的。
- AIXI 需要环境是可计算的，这对于现实世界并不成立。
- AIXI 将智能体与环境视为完全不同的主体，类似于哲学中的二元论。
- 最后，如果我们需实际应用 AIXI，该如何设置奖励？

在笔者看来，奖励的设置尤其微妙。关于 AI 威胁的一个经典故事是：

如果我们将某个 AI 设置为尽可能地制造更多的纸夹（这听上去似乎是一个无害的目标），它就有可能最终穷尽整个星球乃至宇宙的资源，尽可能将一切变为纸夹，并在此过程中毁灭一切其他生命。

这说明了为 AI 设置一个合理目标的重要性。让我们看人类智能是如何解决这一问题的。我们可用强化学习的观点定义人类智能：

人类也可被视为强化学习中的智能体，生存在世界上的目的，在于最大化自己的目标函数。

事实上，在经济学中就是这样定义人类行为：

- 人的目标函数可称为效用 (utility)，一般认为对应于人的“快乐度”，对于每个人不同，在不同的情况下也不同。
- 目前流行的观点是，效用不是基数，是序数。即，只能说“对于这个人，在这个情况下，热狗比苹果更好”，而不能具体说“热狗比苹果好 5 倍”。

但这无法解释为何不同人的目标函数不同。人类的奇妙在于，我们似乎可自行发现目标函数。例如，在生物学上，人的目标是生存和繁衍，但对于喜爱极限运动的人而言，这两者可被抛在脑后。人生没有固定的意义，人可以自己为自己创造意义。有人希望世界美好，有人希望国家强盛，有人希望家庭幸福，有人希望独善其身，也有人自暴自弃。

如果人类最终创造出了达到人类水准的 AI，它的目标函数会是如何？它是否会有能力自行发现目标函数？答案也许在未来会逐渐出现。

9.3.5 创造力、审美与意识之谜

在前文对于生成模型的介绍中，我们看到了 GAN 等方法的威力。

在《CAN: Creative Adversarial Networks, Generating “art” by Learning About Styles and Deviating from Style Norms》^①一文中，研究人员将 GAN 生成的画作与著名当代艺术展 Art Basel 2016 的参展作品，交由人类观众识别，结果是：

- ❑ 观众有 41% 几率认为 Art Basel 2016 的参展作品是人类画作，同时给出了 2.8 的平均分。
- ❑ 观众有 53% 几率认为 GAN 生成的画作是人类画作，同时给出了 3.2 的平均分。换言之，电脑的画作比人类画作“更像人类画作”。
- ❑ 在“喜爱度”“新颖性”“复杂度”等指标上，观众都给 GAN 生成的画作打出了高于 Art Basel 2016 的参展作品的分数。

这是否意味着 AI 已胜过当代艺术家？让我们看看这些作品。首先是 AI 的作品，如图 9-24 所示。在 AI 作品中，最受观众欢迎的作品如图 9-25 所示。

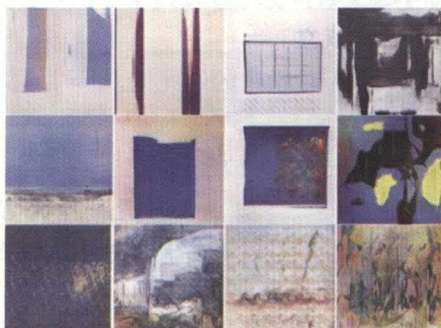


图 9-24 AI 的当代风格画作



图 9-25 受观众欢迎的 AI 画作

而人类参展作品举例如图 9-26 所示。



图 9-26 人类的当代风格画作

^① 地址为 <https://arxiv.org/abs/1706.07068>。

可见，人类的不少参展作品，可能反而更接近于大众心目中的“AI 画作”。这是当代艺术的一种流行风格。

但如果欣赏者有经验，就会看出人类画作的内在逻辑性比 GAN 生成的画作更强。如果熟悉 GAN，也会看出 GAN 的画作中充满着破碎的线条、形体、纹理，具有特殊的挥之不去的“GAN 印记”。

同时，目前 GAN 只擅长抽象画作，很难生成写实风格的作品，因为 GAN 很难精确掌握事物的形体，它画出的效果经不起细看，如图 9-27 所示。



图 9-27 GAN 并不擅长传统画作

AI 还可用于创作其他媒介的艺术，例如 Google DeepMind 的 WaveNet 可自动生成音乐。另一个例子是，微软的 AI “小冰”在学习了大量现代诗后，于 2017 年出版了一本由 AI 生成的诗集《阳光失了玻璃窗》。

在此之前，小冰用匿名投稿的诗作，已成功发表在《北京晨报》《信报》上，以及被《长江诗歌》微信公众号转载。耐人寻味的是：

- 如果不说明是 AI 的作品，那么许多诗歌爱好者和诗人会认为 AI 的诗不错，甚至有些作品比人类诗人的诗更好。
- 但在说明是 AI 的作品之后，许多人就会批判这些诗“没有灵性”“没有感情”“堆砌辞藻”“尚未入门”。

我们不妨来看看小冰的两首诗：

微明的灯影里
我知道她的可爱的土壤
使我的心灵成为俘虏了
我不在我的世界里
街上没有一只灯儿舞了
是最可爱的
你睁开眼睛做起的梦
是你的声音啊

这是一个诗人的教堂
太阳向西方走去我被抛弃
可信的蛇会做云层鱼的声音
听不见声音的天气
若近是语言文字的艺术为自然的国人
待从我的心灵
幸福的人生的逼迫
这就是人类生活的意义

需要指出，在小冰的诗集中，是人从几万篇 AI 作品中挑选出的作品。如果不加挑选，AI 的作品质量会明显更低。

如果严格按照学术标准，这些诗并不能说是 AI 的自行创作，人挑选后的效果是不算数的（令人遗憾的是，许多 AI 论文都会用人挑选，以让效果看上去更好，吸引眼球）。从前网上也有自动作诗程序，偶尔会有佳句（如“高低峭壁临空尽，宽窄长河向地悬”），但这也是人挑选出来的。

也就是说，这是通过人类的审美来挑选。审美就像目标函数，可以指引艺术创作。AI 已经初步学会创作，但 AI 还无法真正学会审美，否则 AI 就能自行挑选出作品，无须人类干预。看来，审美比创作还要更难。正如国内著名科幻作家刘慈欣的《诗云》所述，文中的外星人可成功将人类文字的所有组合存储在一朵云彩中，却无法从中选出佳作。

在某种意义上，AI 创作诗词的过程和许多读者小时候玩过的一种游戏类似，即，随机用主谓宾和定状补，组合出千变万化的句子。因此，AI 作品中的词语搭配更容易出现奇怪的组合，且整体叙述的连贯性会弱于人类。

如何让 AI 学会审美？审美来自何处？来自物理的规律、生物的基因、大脑的运作，还是历史的进程、社会的演变、个人的经历？如果来自多个要素，它们之间如何相互作用？以上是有过诸多争论的极为复杂的问题，也是进行艺术创作的 AI 需要面对的问题。

在笔者看来，目前的生成模型绕过了这些问题，但如果希望生成更高质量的结果，就需要将这些问题思考清楚。审美看似主观，但实际包含高度的逻辑性，正如画家需要学习人体结构与透视，作曲家需要学习和声与配器；人类的艺术创作经过理性思考，而目前的生成模型更像是在表面模仿。其实，让 AI 创作艺术与让 AI 理解语言，有相似之处。

在讨论智能时的终极问题可能是意识，然而我们对于人类意识也知之甚少。它似乎和自我认知、自由意志、时间、记忆、存在感、感官输入、梦等话题有密切联系。

从“笛卡尔剧场”到“缸中之脑”，哲学家对此曾有许多思考。而在科学上，我们仍然无法确定意识来自于大脑的哪个区域（可能是屏状体或前额叶），抑或是大脑作为复杂系统出现的整体现象（也许是量子现象）。

不过，我们可先从其他角度探索意识。在 2017 年 9 月的一篇吸引眼球的论文是《The Consciousness Prior》^①，虽然近年 AI 届有将论文标题越起越大的浮夸风，但这篇论文来自于著名研究者 Bengio，因此它引发了不少讨论。

简而言之，这篇论文认为，意识与对未来的预测息息相关。在智能体中，会存储很多内部变量，例如：

h_t ：一个高维向量，代表智能体对于当前状态的压缩表示

我们可再引入一个低维向量 c_t ，对应于智能体的“意识”的表示：

$$c_t = C(h_t, c_{t-1}, z)$$

^① 地址为 <https://arxiv.org/pdf/1709.08568.pdf>。

其中 z 是一个噪音变量，而 C 称为“意识循环网络”。

我们希望 c_t 包含对于未来状态的预测的信息，换言之，应能构造出另一个神经网络 $V(h_t, c_{t-k})$ ，它能预测在给定 c_{t-k} 后 h_t 发生的概率。我们还可构造出另一个神经网络将 c_t 转变为文字，可能类似于智能体的内心独白（inner monologue）。说起这个话题，推荐对意识和 AI 感兴趣的读者观看 2016 年的美剧《西部世界》，其中有不少有趣的思考。

在这种写法中，意识不再是缥缈不可捉摸的事物，而是一个实实在在的数学定义。笔者也倾向于认为，AI 和通常意义上的意识（不妨在下文称为“真意识”）可能并没有因果关系。而且，如果 AI 看上去像是有意识，在与我们的交互中显得像是有意识，那么它就已在实际的意义上具有意识，即使这种意识是来自于冷冰冰的数学公式。刚才提到的《西部世界》对此有精彩的演绎。

其实，哲学家也早已发现，我们甚至不能判断其他人类是否具有“真意识”（这涉及“哲学僵尸”概念，有兴趣的读者可自行搜索），因此讨论 AI 是否具有“真意识”恐怕也是缺乏建设性的。实际上，许多著名 AI 研究人员都会避免讨论 AI 与意识的关系。

而且，一个强大的 AI 完全可以没有意识。例如，如果一个 AI 能回答人类的各种问题，但没有“自我存在”的概念，这并不会减少这个 AI 的智能程度。

如果 AI 对人类造成威胁，可能并不是因为它像好莱坞大片所描述的那样产生了意识，而是可能来自于程序错误和人类的误用和滥用。因为如果程序出现了错误，一个没有意识的 AI 同样可自发地对人类造成致命的威胁，正如前文提到的“制造纸夹的 AI”。

9.3.6 预测学习：机器学习的前沿

在 NIPS 2016 大会上，著名研究者 LeCun 提出了预测学习（predictive learning）的概念。在他的讲稿中，将机器学习比喻为“蛋糕”：

- 强化学习，是蛋糕上的小樱桃：输入数据，输出 1 个数字，代表对于奖励的预测。
- 有监督学习，是蛋糕的糖霜：输入数据，输出少量结论，例如图像的分类。
- 无监督学习，预测学习，是蛋糕的真正本体：输入数据，输出同样量级的预测。
 - 例如：输入部分缺失的图像，输出将图像补充完整后的结果。
 - 例如：输入一段视频，输出对于视频的未来发展的预测，如图 9-28 所示。

预测学习很重要，而且可能是通往强人工智能的必经之路。目前深度学习的领军人物 Hinton、LeCun、Bengio 均在此投入了研究，也许它将带来 AI 的下一场革命。

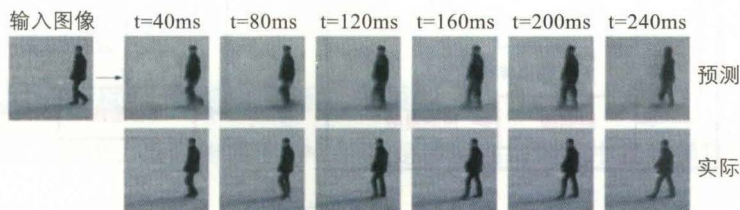


图 9-28 预测学习实例

在笔者看来，预测学习有几个重要特点：

- ❑ 它需要逻辑，需要常识。例如，在预测视频的发展时，需学会物理定律（如牛顿三定律），学会不同事物的特征（如人体的不同关节的运动方法），学会空间立体的概念（需处理物体的平移，旋转，相互间的遮挡），等等。在推理和预测过程中，还需要记忆，包括长期和短期记忆。目前研究人员已有办法为网络加入各种记忆模组。
- ❑ 它的数据特别容易获得。首先，它是无监督方法，无须人工标记。其次，网络上已经有浩如烟海的视频，截取出的每个片段都可用于训练。而且我们还可用摄像头轻松生成无穷无尽的新视频，甚至可让 AI 学会主动控制摄像头。我们还可使用游戏引擎生成视频画面。
- ❑ 此前我们已看到数据对于深度学习的关键性。在使用海量数据训练后，深度网络有可能会逐渐掌握逻辑和常识，改善它此前在这两方面的缺陷，甚至自动发现物理定律。本书第 1 章中提到的“情感神经元”是很好的例证：仅仅是要求网络学会预测，网络就可自动发现深层次的规律。
- ❑ 预测学习也有非常实际的用途。例如，在自动驾驶中，如果能预测其他车辆和行人的行为，就可以改善自动驾驶的性能和安全性。

在 LeCun 的讲稿中，给出了如下公式：

$$\text{智能} = \text{感知} + \text{预测} + \text{记忆} + \text{推理和规划}$$

笔者将其总结为：

$$\text{记忆过去，感知现在，预测未来}$$

如果与强化学习结合，可得到如图 9-29 所示的更完整的架构图。这里智能体的目标是最小化费用，这需要通过对世界建模实现。

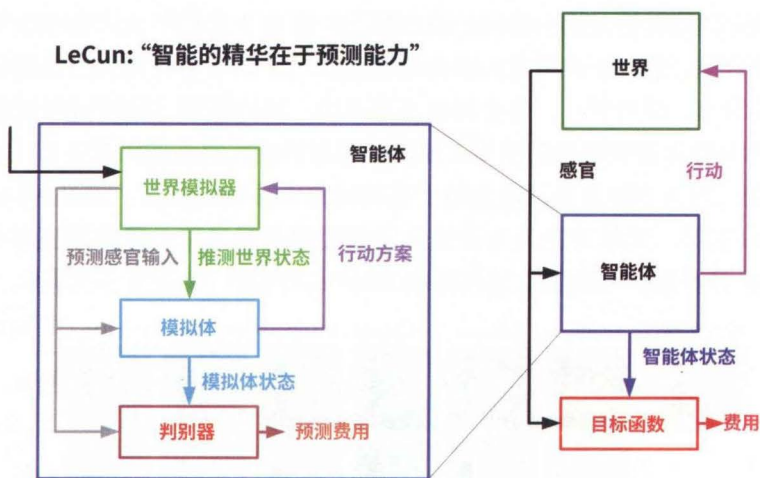


图 9-29 智能体的架构

目前在预测学习领域已有许多有趣的工作，例如让网络预测游戏环境中物体的下落和碰撞轨迹 (PhysNet[⊖])，如图 9-30 所示。

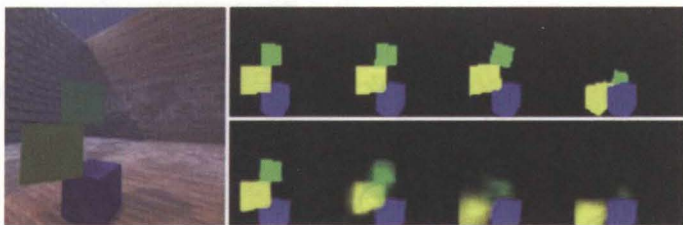


图 9-30 预测游戏画面

左边是游戏场景。右边上排是实际的下落和碰撞情况。右边下排是网络的预测，其中第 1 张图像是网络的输入，后续 3 张图像都是网络的预测。可见，网络有能力大致预测出方块的运动情况，不过会随着时间的推移而越来越模糊。

这是预测模型中的常见现象，因为未来有不确定性。如图 9-31 所示，未来有多种可能性，我们无法提前判知。

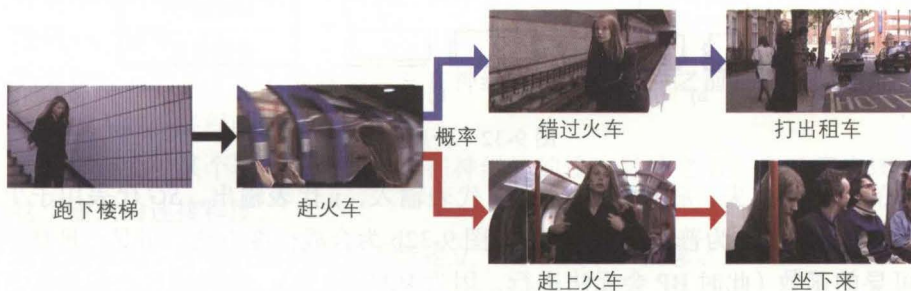


图 9-31 未来的不确定性

因此，如使用 MSE 作为拟合目标，网络会无所适从，最终造成图像模糊。

解决方法是引入 GAN 思想，用判别器网络判断未来的发展是否为真。这样网络可以给出更清晰的预测（虽然预测无法囊括未来的所有发展）。例子见 https://github.com/dyelax/Adversarial_Video_Generation。

9.4 深度学习的理论发展

9.4.1 超越反向传播：预测梯度与生物模型

在过去的几十年中，反向传播 (BP) 一直是神经网络训练中找到梯度的有效方法，但

⊖ 地址为 <https://arxiv.org/abs/1603.01312>。

它并不一定是唯一的方法。

在《Decoupled Neural Interfaces using Synthetic Gradients》^①与《Understanding Synthetic Gradients and Decoupled Neural Interfaces》^②提出 BP 并非必需，因为可用一个辅助神经网络直接拟合出梯度（如前所述，神经网络可拟合任何数据），以实现更灵活的网络训练和网络架构，甚至有可能加速网络训练过程。这称为 DNI（Decoupled Neural Interface）或合成梯度（Synthetic Gradient）技术，如图 9-32 所示。

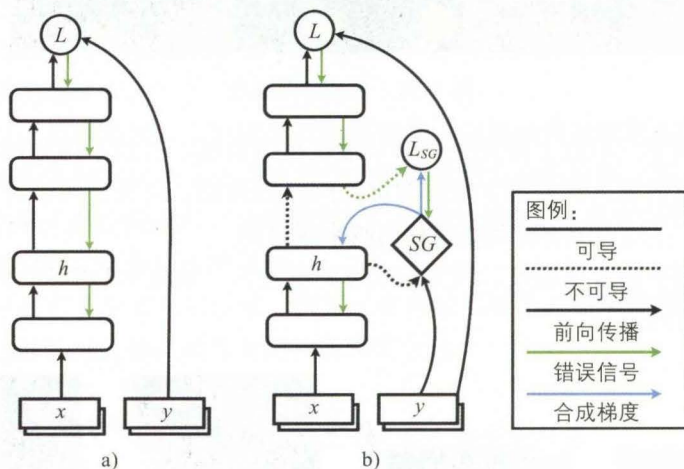


图 9-32 合成梯度

图中的 L 代表损失， h 代表隐藏层， x 代表输入， y 代表输出， SG 代表用于生成梯度的辅助网络。图 9-32a 为普通的 BP 过程，图 9-32b 为合成梯度方法。可见，即使中间层使用了不可导的函数（此时 BP 会无法进行，因为 BP 需求导），也能通过合成梯度方法实现训练。

研究人员还实验了更进一步的想法，即用另一个辅助网络预测前向过程的输入。这类类似于网络的压缩，会略微降低性能。这种想法的意义在于，它可让网络的每一层或每一个模组更加模组化，例如可实现异步训练和异步计算（因为辅助网络会预测网络中其他部分的工作情况）。

类似的研究是《Learning to learn by gradient descent by gradient descent》^③，在其中构造了一个辅助网络，它的输入是另一个网络的当前误差，如图 9-33 所示，目标是希望找到另一个网络的最佳参数更新方法。最终有时可实现比 SGD+BP 更快的训练速度。不过，在训练辅助网络时，还是需要使用 SGD+BP。

① 地址为 <https://arxiv.org/abs/1608.05343>。

② 地址为 <https://arxiv.org/abs/1703.00522>。

③ 地址为 <https://arxiv.org/abs/1606.04474>。

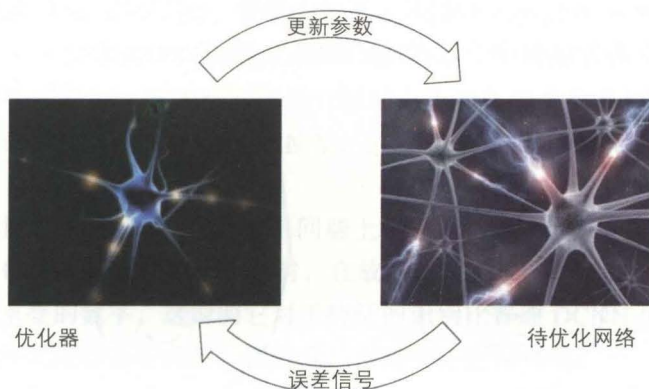


图 9-33 通过辅助网络学习

反向传播的另一大问题是，我们很难想象生物大脑中也有求导和反向传播过程。根据目前的了解，生物神经网络的训练过程遵循 STDP (Spike-Timing-Dependent Plasticity) 机制：

- ❑ 神经元之间通过脉冲电信号通信。如果神经元同时收到来自多个输入的脉冲，就会输出一个脉冲。
- ❑ 如果神经元的某个输入的脉冲经常在神经元的输出脉冲之前一点时间出现，则加强这个输入的连接程度。
- ❑ 如果神经元的某个输入的脉冲经常在神经元的输出脉冲之后一点时间出现，则减弱这个输入的连接程度。
- ❑ 通过此机制，生物神经网络可逐渐建立“因果关系”概念。

通过类似的方法，可在电脑中构造脉冲神经网络 (spiking neural network)，但研究人员尚未找到它的有效训练方法，它的性能不如传统神经网络。

因此，研究人员也会考虑相反的方向，即用生物神经网络模拟电脑神经网络，用 STDP 配合特殊的网络架构模拟 BP 过程，以考察生物神经网络是否实际也是在做 BP。

根据 Bengio 等人的研究^①，这有可能实现。其理论推导过程较为复杂，感兴趣的读者可观看 Bengio 在 CCN 2017 的演讲《Deep learning and Backprop in the Brain》。

9.4.2 超越神经网络：Capsule 与 gcForest

深度卷积网络 DCNN 在计算机视觉领域取得了巨大成功，但“深度学习之父”Hinton 认为，它仍有本质的缺陷难以克服，这些缺陷主要来自于池化层，池化的过程过于简单粗暴（尤其是常用的最大池化），丢失了信息。

① 地址为 <https://arxiv.org/abs/1509.05936>。

举例，如果在图中可找到人的眼睛，鼻子，嘴等，DCNN 就会有大的置信度认为图 中所描绘的是人脸，即使是图中的五官完全错位，如图 9-34 所示。

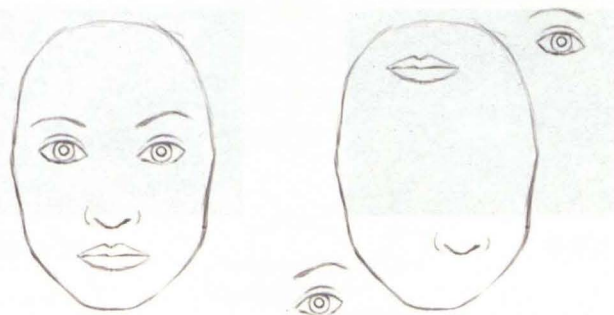


图 9-34 DCNN 的缺陷

在 2017 年 10 月，Hinton 在经过多年研究后，发布了 Capsule 架构^①以探索如何改进 DCNN 的缺陷，并在 AI 界引发了热议。

在 Capsule 架构中，最基本的单元不是单个神经元，而是胶囊 (capsule)。Hinton 希望每个胶囊能学会准确地识别复杂的特征，它的输出不是一个数，而是多个数，即一个矢量，代表特征的各个参数。例如，对于 MNIST 的情况，如图 9-35 所示。包括特征的大小，笔画粗细，宽度，各种变形的情况，等等。

大小和厚度	6 6 6 6 6 6 6 6 6 6 6 6
局部特征	6 6 6 6 6 6 6 6 6 6 6 6
笔画粗细	5 5 5 5 5 5 5 5 5 5 5 5
局部变形	4 4 4 4 4 4 4 4 4 4 4 4
宽度和位置	3 3 3 3 3 3 3 3 3 3 3 3
局部特征	2 2 2 2 2 2 2 2 2 2 2 2

图 9-35 Capsule 在 MNIST 的效果

多组胶囊可相互连接成网络，如图 9-36 所示。

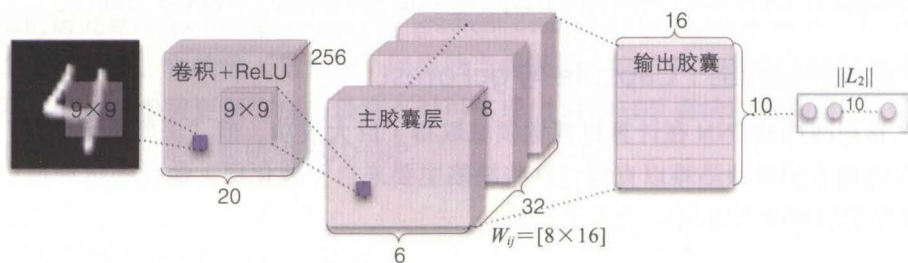


图 9-36 Capsule 网络

① 地址为 <https://arxiv.org/abs/1710.09829>。

训练胶囊网络的方法，不是 BP，而是称为 Dynamic Routing 的方法。在此用简单例子说明其思想。如果一个胶囊负责识别图像中的“左眼”，一个胶囊负责识别“右眼”，那么当两个胶囊输出的矢量接近，就表明左眼和右眼的大小和角度等参数相近，可激活更高层的“双眼”概念。这就实现了向上的逐层抽象，并且整个过程是动态的，相较于池化，更接近我们的直觉。

通过使用胶囊网络，Hinton 在 MNIST 问题上实现了高达 99.75% 的准确率，且在只有少数标签的情况下也能取得较高性能。同时，在故意将 MNIST 中的数字重叠后，胶囊网络也能较好地识别出重叠的数字，这说明它对于特征的识别比普通 DCNN 更为可靠。最后，1 个胶囊也比 1 个神经元更容易被理解，因为它识别的特征更为明确。

另一方面，根据 https://github.com/jaesik817/adv_attack_capsnet 的测试，胶囊网络对于对抗样本的抵抗力确实更高，不过仍然可被骗过。此外，在更复杂和更接近现实的 CIFAR-10 数据集中，胶囊网络目前只能实现 10.6% 的错误率。或许未来的研究人员可找到继续改进其性能的方法。

在 2017 年出现的另一 DCNN 挑战者是 gcForest[⊖]，来自我国机器学习界的领军人物，南京大学的周志华教授，如图 9-37 所示。

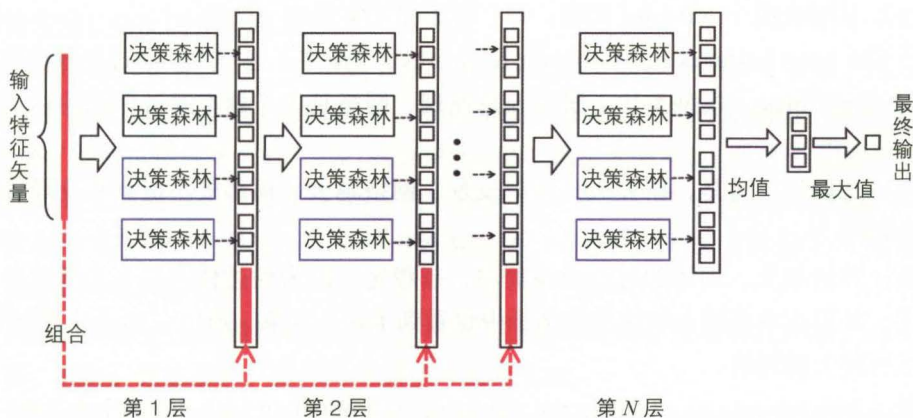


图 9-37 gcForest 架构

gcForest 也是一个层次模型，但每层使用的不是神经元，而是由决策树构成的森林。其中也使用了类似残差结构的思想，将原图与每层的输出组合，作为下一层的输入。它在 MNIST 数据集上能达到 99.26% 的准确率，且训练所需的超参数比 DCNN 更少，不过它对于 CIFAR-10 数据集的性能也不尽如人意。

这从侧面说明了 DCNN 的威力。在简单数据集和数据较少的情况下，DCNN 的性能并不一定最好。但在复杂数据集和数据充足的情况下，确实往往很难找到性能胜过 DCNN 的模型。

⊖ 地址为 <https://arxiv.org/abs/1702.08835>。

另一方面，在 AI 的历史上，最佳的模型会不断改变，或许未来我们会找到全面胜过 DCNN 的新方法，而这需要我们将部分精力投入开辟更多的新领域，而不是所有人都只在 DCNN 上不断挖掘。

9.4.3 泛化问题

深度网络 DNN 的泛化性能为何如此之强，是深度学习理论中的一大难题。对此的研究有 2 个层次：

- 第 1 层次，研究为何 DNN 能拟合复杂的训练数据，而不会陷入局部极小值点（local minimum）。这属于数学问题，相对简单。
- 第 2 层次，研究为何 DNN 能在实际测试数据中实现良好的泛化性能。这与实际测试数据背后的规律有关，难度更大。

这里的第 1 层次，涉及的概念是损失平面（loss surface），即损失函数（loss function）随参数的变化情况。

训练网络的目标是最小化损失函数。从前的看法是，通过梯度下降方法在大多数情况下只能到达局部极小值点（local minimum）。如果希望到达全局极小值点（global minimum），往往会是个 NP-hard 问题，即，很难保证实现这一目标。

而《The Loss Surfaces of Multilayer Networks》^①证实了这一点。通过与理论物理中的自旋玻璃（spin glass）模型类比，研究人员发现，网络越大，越难通过训练到达全局极小值点。

但是，这里的关键是，研究人员同时发现，网络越大，局部极小值点和全局极小值点的差距会越小。

因此，网络越大，训练的过程会越简单，越稳定，因为到达任意一个局部极小值点就足够好了。我们在本书第 2 章结尾的例子中也看到了这一现象。所以，如果计算资源充分，应使用尽可能大的网络。

根据《Identifying and attacking the saddle point problem in high-dimensional non-convex optimization》^②，真正阻碍收敛的往往是鞍点（saddle point），而不是局部极小值点。本书第 3 章也曾对此介绍。

对于第 2 层次，在 2016 年底的著名论文是《Understanding deep learning requires rethinking generalization》^③。研究人员在实验中发现，即使将训练数据的标签和图像设置为噪音，深度网络也会若无其事地完全学会所有训练数据（此时就相当于完全死记硬背，因为标签和图像是随机设置的没有意义的数字），如图 9-38 所示。

① 地址为 <https://arxiv.org/abs/1412.0233>。

② 地址为 <https://arxiv.org/abs/1406.2572>。

③ 地址为 <https://arxiv.org/abs/1611.03530>。

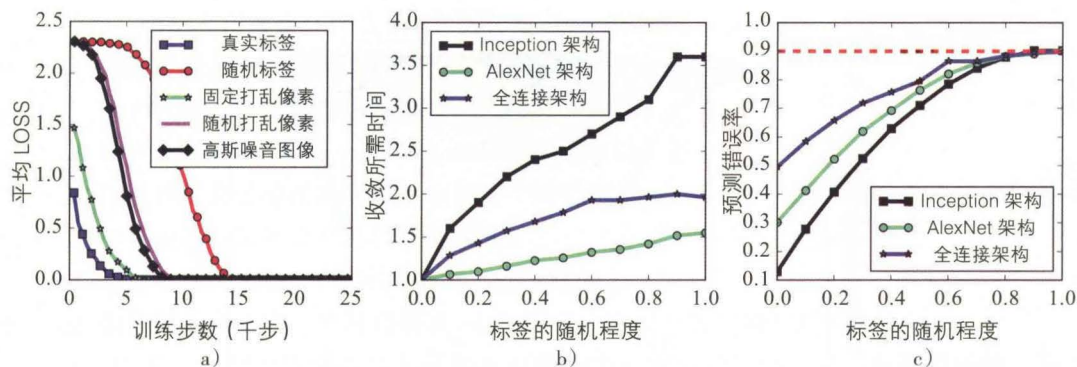


图 9-38 深度网络可以学会随机数据

如图 9-38a 所示，使用随机标签（红色），将图像像素按某固定的乱序打乱（绿色），随机打乱（粉红色），使用完全随机的图像（黑色），最终的训练误差都可降到 0。

如图 9-38b 所示，随着标签的随机程度增加，训练速度会越来越慢，但不同架构的变慢程度与架构的性能并没有关联性。

如图 9-38c 所示，随着标签的随机程度增加，网络的预测错误率也会随之增加，但不同架构的错误增长情况同样是相近的。

此外，研究人员还实验了使用核方法（kernel trick）的线性模型（相当于浅层神经网络），它可直接通过计算实现 0 训练误差，我们希望了解它的泛化性能如何。研究人员发现，它在 MNIST 的测试错误率只有 1.2%（这证明 MNIST 确实太简单）。而对于 CIFAR-10 问题，如果预先做高斯模糊处理，可实现 46% 的错误率；如果预先用 32 000 个随机卷积核处理，错误率就会降到 15%（注意这还没有使用数据增强，否则训练时的计算量太大），已经接近早期深度神经网络的性能。

这说明，如果使用合理的训练方法，并加入图像预处理，浅层神经网络的泛化能力实际也不错。或许，这也说明，图像分类其实没那么难。

在 2017 年的另一篇热议论文是《Opening the Black Box of Deep Neural Networks via Information》^①。它发现，网络的训练可分为两个阶段：

- 第一阶段是“拟合”。即网络逐渐学会输入数据。这个过程的学名，是“经验风险最小化”（Empirical Risk Minimization, ERM）。
- 第二阶段是“压缩”。即网络逐渐脱离输入数据，找到更核心的规律，实现泛化。

如图 9-39 所示，网络的高层（层 5）在一开始与数据和标签的信息相关度都很低。随着训练的进行，它与标签的信息相关度不断提高（因为最终需要给出标签），而与数据的信息相关度先提高（从 A 到 C）后降低（从 C 到 E），说明它一开始是着眼于输入数据，后来就逐渐脱离了输入数据，转为关注更高层次的规律。

① 地址为 <https://arxiv.org/abs/1703.00810>。

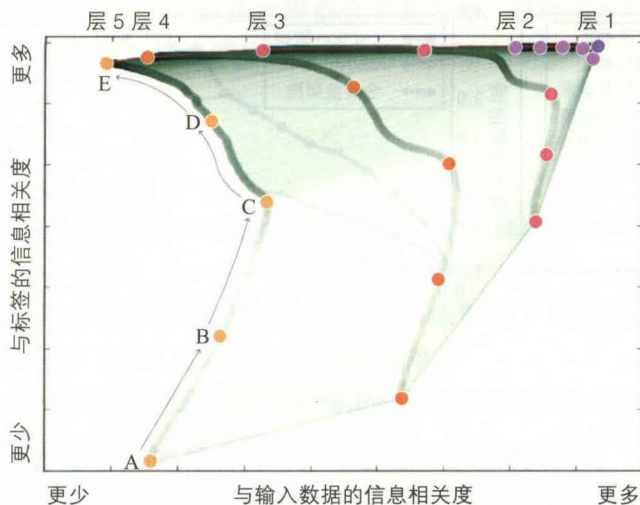


图 9-39 网络训练中的信息流动

这听上去很有趣，就像我们平时说的“先把书读厚，再把书读薄”，引来了许多科技媒体争相报道。Hinton 也表示，这是近年来少有的突破，是真正原创的想法。

不过，在本书写作的时候，也有研究人员认为这篇文章有问题，因为其中的网络架构采用的是全连接网络，而且使用 tanh 和 sigmoid 激活，而非 ReLU 激活。经过实验，使用 ReLU 激活后，这些有趣的现象似乎就消失了^①。这并不代表该研究没有价值，只是说明事情可能没这么简单，需要用更多的模型检验。

说起这个问题，目前部分 AI 论文的作者会回避对于自己不利的实验结果，这点值得读者注意。例如，如果某个模型只发布了在简单数据集上的性能（如 MNIST），那可能是因为这个模型在更复杂数据集（如 CIFAR-10）的性能不佳，论文的作者不好意思说出来。

此外，目前也有研究人员试图从物理的角度解释深度学习的有效性，有能力的读者可参阅《Why does deep and cheap learning work so well?》^②。

简单地说，机器学习希望学会统计分布，统计分布在物理上来自哈密顿量（Hamiltonian）。如果哈密顿量具有简单的形式（例如，是阶数低的多项式），或明确的对称性，或存在简单的有效理论（effective theory），就会适合于神经网络的学习。此外，我们宇宙存在明显的层次结构（hierarchical structure），这也很适合于深度网络的训练和运作。不过，这仍然是停留在想法层面，研究人员仍未找到严格的理论模型。

最后，在 2017 年 12 月的一篇有趣的文章是《Deep Image Prior》^③。它让一个深度卷积网络去学习复制被破坏的图像（如，加入噪点的图像），然后发现这个网络会自动先学会重建图像。

① 地址为 https://openreview.net/forum?id=ry_WPG-A-¬eId=ry_WPG-A-。

② 地址为 <https://arxiv.org/pdf/1608.08225>。

③ 地址为 https://dmitryulyanov.github.io/deep_image_prior。

例如，给定一幅被破坏的图像 x ，具体过程如下：

- 用随机参数初始化深度卷积网络 f 。
- 令 f 的输入为固定的随机编码 z 。
- 令 f 的目标为：输入 z ，输出 x 。以此训练 f 的参数。
- 注意选择合适的损失函数。例如对于降噪问题可关注整体的 MSE，对于填充问题就应只关注不需要填充的位置的 MSE。
- 当训练很久之后， f 可实现输出一模一样的 x 。
- 但如果在训练到一半时打断 f ，会发现它会输出一幅“修复过的 x ”。

这意味着，深度卷积网络先天就拥有一种能力：它会先学会 x 中“未被破坏的，符合自然规律的部分”，然后才会学会 x 中“被破坏的部分”。例如，它会先学会复制出一张没有噪点的 x ，然后才会学会复制出一张有噪点的 x 。

换言之，深度卷积网络，先天就了解自然的图像应该是怎样的。这无疑来自于卷积的不变性和逐层抽象的结构。

Deep Image Prior 的重要特点是，网络由始至终，仅使用了被破坏过的图像 x 作为训练。网络没有看过任何其他图像，也没有看过正常的图像，但最终效果依然颇为不错。这说明自然图像的局部规律和自相似性确实很强。

下面看图 9-40。最左边是带瑕疵的图像 x ，它经过 JPEG 压缩，有很多压缩瑕疵。网络的目标是学会输出它。在 100 次迭代后，网络学会了输出模糊的形体。在 2400 次迭代后，网络学会了输出清晰光滑的高质量图像。在 50 000 次迭代后，网络才学会了输出带瑕疵的图像 x 。



图 9-40 用 Deep Image Prior 修复图像

下面的图 9-41 是将图像由部分像素重建。可见，深度卷积网络很擅长处理不断重复的纹理，效果比传统方法更佳。

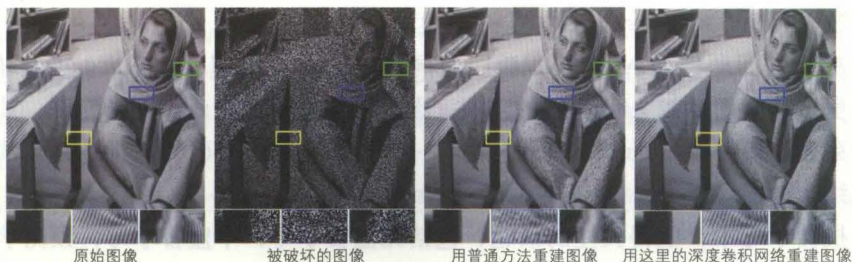


图 9-41 用 Deep Image Prior 重建图像

9.5 深度学习与人工智能的展望

预测未来对于 AI 很难，对于人类也同样难。在本书正文的最后，笔者尝试对深度学习与 AI 的未来发展作出预测。

9.5.1 工程层面

在软件与硬件工程的意义上，目前的深度学习仍处于早期阶段。笔者预期这会逐渐得到改善：

- ❑ 深度神经网络将更易用、更模块化、更灵活、更独立、更可重用，成为类似库函数的概念。我们将更多地使用预训练好的网络，通过将其组合，或迁移学习，实现我们的目标，无须每次费时费力从头训练。目前典型的例子是图像分类中的 VGG 网络，它常常被应用于其他领域的问题。
- ❑ 由于使用深度神经网络实际并不需要很多专业知识，深度学习可能将如普通编程一样成为日常业务。更多的程序员将学习和应用深度学习。
- ❑ 同时，随着深度学习技术的普及，目前业界的泡沫化情况将会减少。正如从前 iOS 和 Android 编程是高薪职业，但随着技术的普及，它们的中低端职位的薪水就会明显减少。
- ❑ 深度学习将更移动化，分布化。目前最新的 iOS 和 Android 设备中都已整合了深度学习芯片，并已出现各种移动设备专用的深度学习库，如 TensorFlow Lite。通过聚集更多的数据和算力，可训练出更强大的网络。
- ❑ 正如本书第 1 章的 AI 历史所显示，AI 的发展与计算硬件的发展密不可分。深度学习专用硬件将逐步涌现和普及。更多设备中将嵌入深度学习芯片，这将与物联网相结合，实现更智能化的生活，更智能化的城市，更智能化的社会。

9.5.2 理论层面

在前文我们曾讨论过 AI 的理论发展，笔者的预测是：

- ❑ 深度学习仍将是 AI 的中心方法，但会与逻辑推理系统、知识图谱等有更多结合。在结合逻辑后，深度神经网络将不再可微，因此不能简单地做 BP，也许需要进化算法 (evolution algorithm) 或合成梯度方法。
- ❑ 解决语言问题将是 AI 的里程碑。这同样需与逻辑推理、知识库结合。有趣的是，这里可能会用类似 AlphaGo 的方法，因为需要进行搜索。以翻译为例，其中需要进行束搜索 (beam search) 以最大化某一分数。目前已有研究人员在此尝试加入策略网络。
- ❑ 在机器学习中，值得关注的话题，包括无监督学习，半监督学习，预测学习，对关系化数据的学习，强化学习，迁移学习，单次学习，连续学习，等等。人类大脑

只有一个巨型神经网络，但可处理多领域的复杂问题，且每时每刻都在自我学习之中。这值得 AI 借鉴。

- 目前我们在深度网络的架构上投入了很多研究，但随着算力的增加，完全可让深度网络自行实验各种网络架构，自行发现更好的架构。深度网络也应能自行发现最佳的超参数，毋需由人设定。这可通过进化算法实现，目前也已出现了这方面的研究。目前主要的障碍是，这对于算力的需求很高，因为需要做大量实验，所以现在只有大公司才能负担。
- 我们需要更好地理解深度网络的运作，从网络中提炼出（distill）更符合人类认知模式的知识，构建可解释的 AI。这将有助于人类和 AI 的共同进步。
- 与此相关的是，需要构建可验证的 AI，难以被欺骗的 AI，以适用于医疗、驾驶等关键领域。

9.5.3 应用层面

前文介绍了许多有趣的深度学习与 AI 实例，但其中仍少有真正落地，能带来营收的应用。这也许将逐渐改变：

- AI 的目的，首先应是代替人类的机械性体力和脑力劳动，将我们从简单的重复工作中解放，提高我们的生活水准。正如工业机器人已逐渐取代制造业工人，扫地机器人已进入许多家庭。
- 在面向公司（2B）的层面，机器学习方法已在推荐系统、人脸识别、客服机器人等领域有较为成熟的应用。未来 AI 将可能在研发、运营、决策、金融层面发挥更多作用，实现更自动化、精细化的管理，提升公司的竞争力。
- 在面向消费者（2C）的层面，目前 AI 的常见应用是，照片搜索和美化，语音助手和翻译，但它们均属于锦上添花的性质。未来真正有可能首先带来社会变革的技术是自动驾驶。
- 教育与医疗均是 AI 可大有作为的领域。基于机器学习的方法可更好地了解每位学生的水平，并因材施教。
- 艺术和娱乐领域同样值得关注。AI 可辅助艺术创作（如最新版 Photoshop 已加入了许多 AI 技术），也可直接生成作品。
- AI 将是所有学科的交叉学科，因为 AI 希望自动完成所有学科的工作。笔者预测，AI 技术将进入各个学科，实现 AI+ 数学，AI+ 物理，AI+ 化学，AI+ 生物，AI+ 医学，AI+ 工程，AI+ 材料，AI+ 经济，AI+ 金融，AI+ 文学，AI+ 艺术，AI+ 历史，AI+ 教育，AI+ 管理，等等。
- AI 与机器人的结合将让我们的生活更便利，但如前文所述，也可能带来威胁。

人工智能与我们的未来

在 2012 到 2013 年期间，Bostrom 向数百位 AI 专家发送问卷，预测“能够在大多数职业上达到典型人类的水平”的强人工智能会在何时实现，调查结果汇总如下：

- ❑ 乐观估计（有 10% 的可能在这一年达成）：2022 年。
- ❑ 正常估计（有 50% 的可能在这一年达成）：2040 年。
- ❑ 悲观估计（有 90% 的可能在这一年达成）：2075 年。

这一调查结果非常稳定，在 2007 到 2013 年的多次调查中，预测的中位年都在 2040 年左右。

而 2016 到 2017 年的最新调查[⊖]显示，研究人员对于“能够在所有任务上比人类更好更便宜地完成任务”的超人工智能的中位预测是 2061 年。

亚洲与北美研究者（以毕业院校分类）在此的分歧很大。中国研究者明显更乐观，中位年预测是 2044 年。美国研究者明显更谨慎，中位年预测是 2092 年。

这可能是因为调查中的中国研究者更为年轻。分析显示，年轻的研究者更乐观，资历较浅的研究者更乐观，男性研究者比女性研究者更乐观。

研究人员预测，卡车司机、售货员等工作将在十余年内被 AI 取代；且 AI 将在十余年内写出登上排行榜的流行歌曲，三十余年内写出《纽约时报》排名中的最畅销书，四十余年内成为顶尖的数学研究者，八十余年内成为顶尖的 AI 研究者，实现完全脱离人类的自我完善和进化。

如果预测能成为现实，那么 AI 将对人类社会带来前所未有的影响：

- ❑ 可将 AI 类比于历史上的“电力”和“互联网”，它有能力显著提高我们的生活水准，让许多旧的职业消失，并同时创造新的职业，解放生产力。

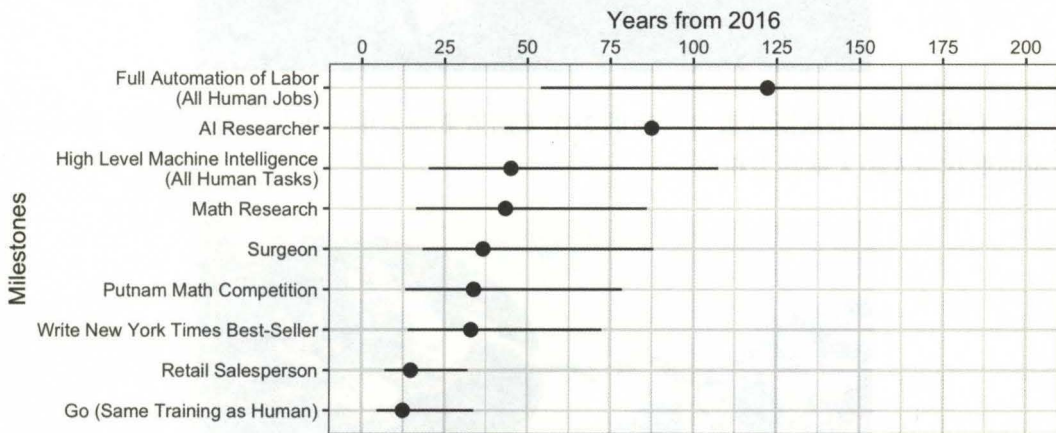
⊖ 地址为 <https://arxiv.org/abs/1705.08807>。

- 可以想象，未来所有移动设备，电器，公共设施，车辆等将全部联网，AI 可获得取之不尽的数据，全面优化社会的运作。AI 将在多个职业代替人类。人类的公司、组织、政府、军队等将在越来越大的程度上由 AI 负责精确管理。
- 但 AI 与此前技术的区别在于，它可能会逐渐取代人类的一切职业。这可能会让我们享受美好的生活（一切事情都有机器为我们服务），也可能会带来大规模的失业潮，造成贫富差距进一步扩大，与人类的迷失和边缘化，这是值得注意和警惕的。
- 由于 AI 对于数据和算力的巨大需求，目前 AI 技术已越来越集中在少数巨型公司中。这种技术垄断的趋势，不利于社会发展，也令许多业界人士表示担忧。目前已出现 OpenAI 等机构致力于让 AI 的研究更加开放，但更重要（和更困难）的是促进数据和算力的共享化，分布化。

而 AI 本身是否会对人类带来威胁，在笔者看来，答案可能是否定的：

- 在科幻电影中常出现的是 AI 产生意识然后攻击人类，如《终结者》系列中的 Skynet 天网（在电影中 Skynet 正是基于神经网络）。这实际可能性很低。更有可能的是 AI 的 bug 导致事故，或 AI 被极端组织利用，或 AI 导致更致命的军事冲突。
- 一种保证 AI 安全性的方法是构建 Oracle（万能回答者）型 AI，它可回答人类的任何问题，但不参与最终的决策和执行，也不存在自我的动机。

说起这个科幻性质的话题，在好莱坞大片中 AI 攻击人类的理由，往往是出于对地球的有限资源的争夺。不过，在笔者看来，如果人类真的有能力创造出具有自我意识的 AI，届时我们可能已有能力进行星际旅行。宇宙如此浩瀚，也许不必再局限于地球。即使届时人类尚未拥有这种能力，AI 或许也会想去探索更广阔无垠的宇宙，而不是在小小的地球上与人类争斗。

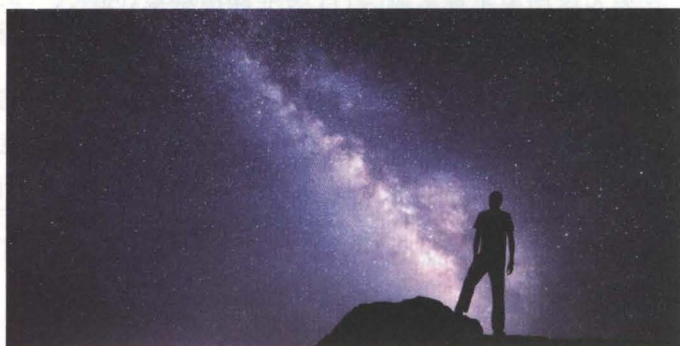


在笔者看来，人与 AI 的关系，还可以有新的道路，因为人类智能也并未达到极限，人类智能与 AI 有可能相辅相成。与 AI 相类比，我们也可将人类智能分为软件的部分和硬件的部分：

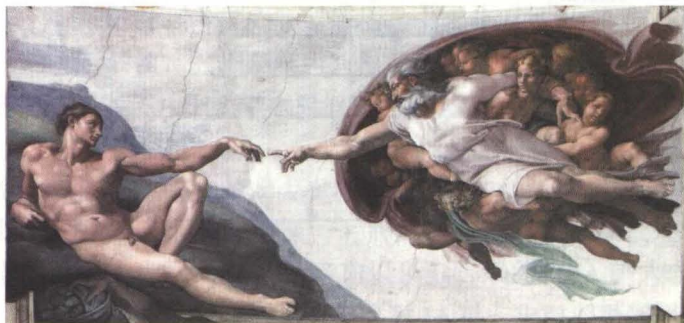
- 对于人脑的软件，新技术将全面提升人类的教育水准和信息交流速度，提升人类的群体智慧。正如互联网的出现，让新研究进展可在瞬间传遍世界（如 <http://arxiv.org/>）。同时，AI 可将教育视为一个优化问题，找到适合每个人的激发潜力的方法，让人类更充分发挥自身的价值。
- 对于人脑的硬件，我们可通过基因工程，纳米机器人等方式增强我们的大脑和身体，也可通过脑机接口、芯片植入、思维上载等方式实现与 AI 和机器的更紧密结合，增强（augment）人类。正如柯洁在人机大战后表示“如果有一天，科技已经能让我植入芯片，我也会义不容辞地植入芯片，我也会去报仇”。
- 最终，人类将有可能与 AI 相融合，成为一种全新的生命形态。

在 AI 如火如荼的今天，我们对于人类智能的研究和投入相对远远更少。但顶尖 AI 研究人员已达成共识，AI 的继续发展离不开对于人类智能的理解。他们已致力于对人类智能和人脑的机理进行更深入的研究。

对于此感兴趣的读者，可关注 Hinton、Lecun、Bengio 等 AI 领军人物的研究动向，以及 <https://www.goodai.com/>，<https://www.roadmapinstitute.org/> 等机构。



在米开朗基罗于西斯廷教堂穹顶的杰作《创造亚当》中，医学家曾注意到，画中红色裹布的形状与人类大脑甚为相似。



人类的生物大脑，带领我们在 300 万年前凿出石器，6 万年前发明弓箭，5000 年前建

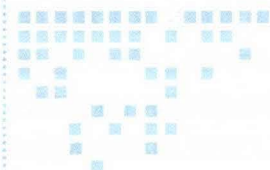
立国家，200 年前工业革命，50 年前登上月球。无数先哲都曾困惑于我们的智能来自何方，如何运作，我们在宇宙中是否唯一。

而在 300 万年后的今天，我们隐隐已在扮演着造物主的身份，逐渐创造一种新的智能形态。AI 有智能，但 AI 不一定有意识，不一定有生命。如果 AI 没有人类的血肉之躯，没有人类的生老病死，那么 AI 就难以真正与人类产生共情，难以体会人类的喜乐与哀愁。然而这一切可能都不重要。宇宙中如果还有其他智能形态，也可能会与我们有天壤之别。

通向智能之秘的道路，也必然会有更多曲折与挑战，有待我们砥砺前行。深度神经网络或许不是终点，而是下一次 AI 飞跃的契机。谨以本书，献给所有志在研究、理解与揭示智能奥秘的读者。

彭博

2018 年 1 月 18 日于深圳



Appendix

附录

深度学习与 AI 的网络资源

除下列资源外，知乎 (<https://www.zhihu.com>) 是目前中文网络上较为专业的交流平台，读者可在其中找到深度学习和 AI 的介绍和综述，对于最新论文的分析，以及问题解答。

读者可扫码关注作者的知乎主页 (<https://www.zhihu.com/people/bopengbopeng>)，其中定期会有 AI 和相关的文章。



1) 深度学习与 AI 的资源列表

中文列表，包括教程和科普，以及经典论文：

<https://github.com/ty4z2008/Qix/blob/master/dl.md>

<https://github.com/ty4z2008/Qix/blob/master/dl2.md>

英文列表，包括大量论文、教程、代码，数据库等：

<https://github.com/endymecy/awesome-deeplearning-resources>

<https://github.com/ChristosChristofidis/awesome-deep-learning>

深度学习的最新论文:

<http://www.arxiv-sanity.com/>

国外著名论坛 Reddit 的机器学习讨论区:

<https://www.reddit.com/r/MachineLearning/>

经典论文大全:

<https://github.com/terryum/awesome-deep-learning-papers>

<https://github.com/songrotek/Deep-Learning-Papers-Reading-Roadmap>

GAN 论文大全:

<https://github.com/zhangqianhui/AdversarialNetsPapers>

<https://github.com/nightrome/really-awesome-gan>

2) 经典教程

Stanford 大学深度卷积网络教程:

<http://cs231n.stanford.edu>

神经网络入门教程:

<http://neuralnetworksanddeeplearning.com>

业界专家吴恩达 (Andrew Ng) 的视频教程:

<https://www.coursera.org/specializations/deep-learning>

面向程序员的综合教程:

<http://course.fast.ai/>

<http://course.fast.ai/part2.html>

3) 经典数据集和应用

机器学习的数据库大全:

<https://github.com/caesar0301/awesome-public-datasets>

图像识别的世界记录演变, 相关的经典论文, 及数据库下载:

http://rodrigob.github.io/are_we_there_yet/build/

深度学习在 CV 中的经典应用:

<https://github.com/kjw0612/awesome-deep-vision>

GAN 模型大全:

<https://github.com/hindupuravinash/the-gan-zoo>

<https://github.com/hwalsuklee/tensorflow-generative-model-collections>

正在复现 AlphaGo Zero 的开源围棋项目 Leela-zero:

<https://github.com/gcp/leela-zero>

棋力强的开源围棋程序 AQ:

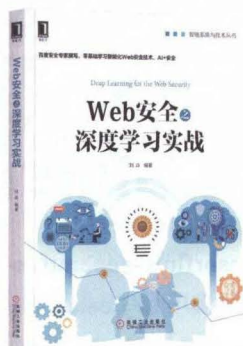
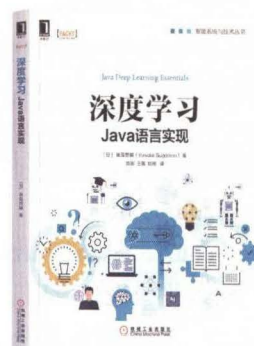
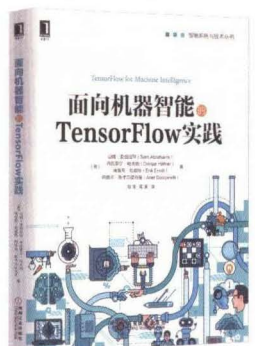
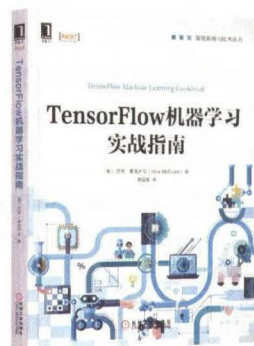
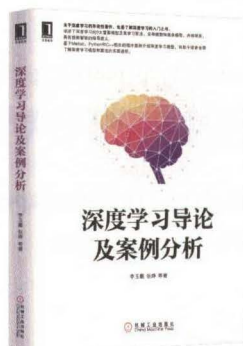
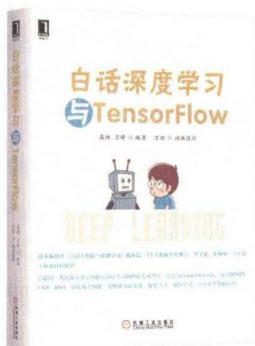
<https://github.com/ymgaq/AQ>

棋力强的开源围棋程序 RN:

<https://github.com/zakki/Ray>

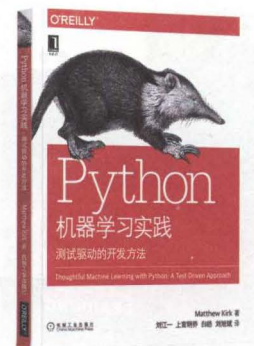
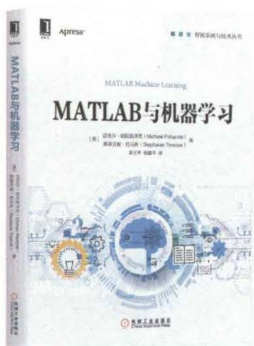
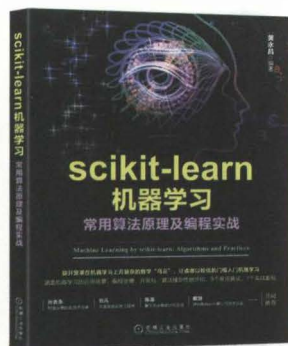
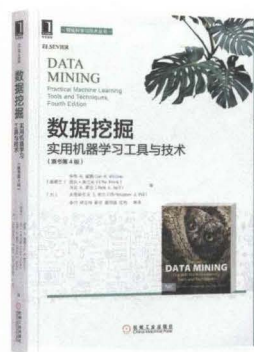
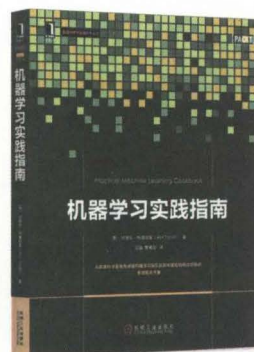
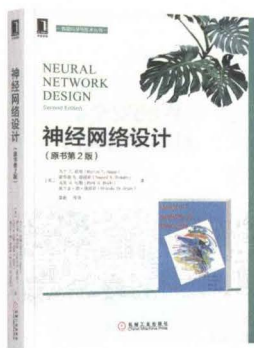
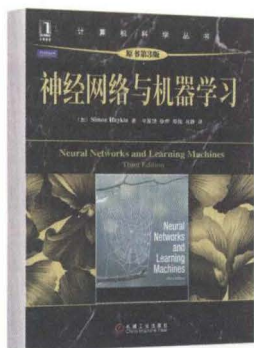
推荐阅读

深度学习系列



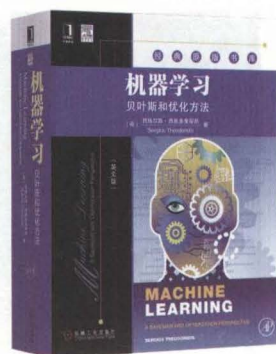
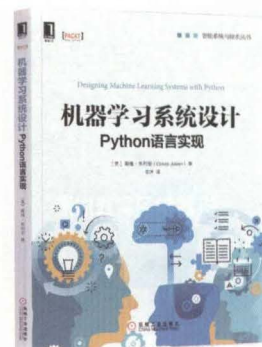
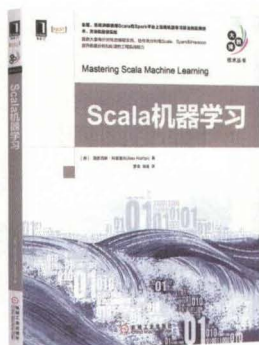
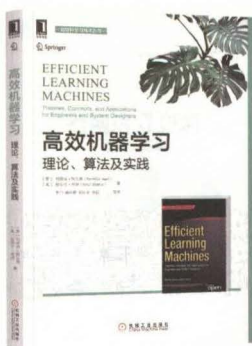
推荐阅读

机器学习与神经网络系列



推荐阅读

机器学习系列



作者简介

彭博 人工智能、量化交易、区块链领域的资深技术专家，有20年以上的研发经验，智能硬件公司稟临科技（withablink.com）的联合创始人。

在人工智能与信息科技方面，对深度学习、机器学习、计算机图形学、智能硬件等有较为深入的研究。

在量化交易方面，曾负责全球最大的外汇对冲基金的程序化交易，对市场的微观和宏观行为有较为深入的理解。

在区块链方面，对智能合约、DApp开发和自动交易有较为深入的实践。

知乎上科技领域的大V，撰有大量技术文章。

Deep Convolutional Neural Network

Principle and Practice

2016年AlphaGo人机大战开启了人工智能的黄金时代，CSDN数据统计表明，2016年深度学习技术文章全年增长3倍以上，阅读过机器学习和深度学习内容的开发者近百万，但全面掌握深度学习还是有门槛的，坊间也有深度学习的经典著作，但缺乏对系统工程实践方面的讲解，彭博的这本书弥补了这一缺失。本书由浅入深，从实践出发，剖析AlphaGo的实现，全面、系统地介绍了深度卷积网络的应用和发展，开发者跟着书中案例一步步实现，就可以快速掌握深度学习的精要。

—— 蒋涛 CSDN创始人/极客邦创投创始人

人工智能是未来重要的发展方向，也是我们极客邦最重要的内容版块之一。这本书选择深度学习中最受欢迎的技术——深度卷积网络（DCNN）和生成式对抗网络（GAN）为切入点，从技术、原理、训练方法、应用等角度对它们进行了全方位讲解。为了让艰深的技术不那么难懂，书中以我们熟知的AlphaGo为案例（AlphaGo是使用深度卷积网络的经典成功案例），通过分析AlphaGo的实现过程，让具体的技术细节通俗易懂。

—— 霍泰稳 极客邦科技创始人兼 CEO

神经网络是深度学习最有效的方法之一，深度卷积网络是目前十分流行的深度神经网络架构，在图像、视频、语音等领域被广泛应用，我们熟知的AlphaGo就大量使用了深度卷积网络技术。彭博的这本书，从理论和实践两个层面对卷积神经网络做了非常翔实的讲解。理论层面，重点讲解了卷积神经网络的技术基础和工作原理；实践层面，则通过对AlphaGo的层层剖析，展现了卷积神经网络的实践方法。此外，本书在大环境上，基于Python和MXNet等流行的深度学习工具撰写，能让读者在理解深度卷积网络的同时，掌握这些工具的使用。

—— 刘付强 MSUP（麦思博）创始人兼CEO

对于初次接触人工智能的工程师和数据科学家来说，虽然系统性的建立对人工智能知识体系很重要，但同时也需要找一个易于学习的切入点或突破点。本书作者考虑到不同层次读者需要，选择了很有代表性的AlphaGo和GAN进行讲解，并给出了很好的案例引导。本书循序渐进地从概念到原理，从原理到实践，步步深入AI，案例生动而细致。持续学习是人工智能发展的关键，同样，在这样一个技术日新月异的时代，我们也需要保持持续的学习力。推荐本书，保有一份对技术的敬畏与热忱，让我们一起打开AI的世界吧。

—— 梁勇 天善智能创始人



上架指导：计算机/机器学习

ISBN 978-7-111-59665-3



9 787111 596653 >

定价：129.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn